Combining a state based formalism with temporal logic

Thomas Deiß

deiss@informatik.uni-kl.de

05/1996

Sonderforschungsbereich 501

Universität Kaiserslautern
Erwin-Schrödinger Straße
D-67653 Kaiserslautern

Combining a state based formalism with temporal logic

Thomas Deiß*
Fachbereich Informatik, Universität Kaiserslautern
67663 Kaiserslautern, Germany
deiss@informatik.uni-kl.de

Abstract

A combination of a state-based formalism and a temporal logic is proposed to get an expressive language for various descriptions of reactive systems. Thereby it is possible to use a model as well as a property oriented specification style in one description. The descriptions considered here are those of the environment, the specification, and the design of a reactive system. It is possible to express e.g. the requirements of a reactive system by states and transitions between them together with further temporal formulas restricting the behaviors of the statecharts. It is shown, how this combined formalism can be used: The specification of a small example is given and a designed controller is proven correct with respect to this specification. The combination of the langugages is based on giving a temporal semantics of a state-based formalism (statecharts) using a temporal logic (TLA).

1 Introduction

It is considered generally a difficult task to capture the requirements of a system which is behaving dynamically. To avoid misunderstandings of the requirements, these should be stated in a manner, which is as precise as possible. On the other hand, such a document has to be understandable by the customer, which typically has a different background.

This paper is concerned with convenient formalisms for descriptions of reactive systems. In *reactive systems* only the relative order of events with respect to time is of interest. Neither metric aspects of time nor further continuous variables are considered[1]. To describe such a system it will be necessary to talk about the environment in which it is operating. Also, desired properties of the systems have to be stated and at last a description of the reactive system itself has to be given. A lot of languages for these descriptions are existing already, each of them having its specific advantages and disadvantages. To use the benefits of model- as well as of property-oriented specification styles and the corresponding languages, a combined language – to be used for each of these descriptions – is defined.

This combined language is based on statecharts [Harel, 1987]. Given a statechart, it is shown, how a temporal formula – in TLA [Lamport, 1994b] – can be defined, which is satisfied by the

*This research was supported by the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 501 "Entwicklung großer Systeme mit generischen Methoden".

[1]This classification of reactive systems is taken from [Manna and Pnueli, 1993].

same behaviors as the statechart. It is easily possible to express specific properties in temporal logic by conjoining further formulae to the one capturing the behaviors of the statechart.

By a small example it is shown, how this combined language can be used for the descriptions mentioned above. All descriptions are based on the that of the environment or plant, which thereby plays a central role among the descriptions. Relationships between the various descriptions are considered within the temporal framework of TLA, which is usable because the semantics of a statechart has been defined within TLA.

The paper starts with some general considerations about descriptions of reactive systems. It is followed by a short introduction of TLA, including syntactical conventions, in section 3. Section 4 is used to define the temporal semantics of (a subset of) statecharts. The rest of the paper presents the descriptions of the example – a simple lift controller. The description of the plant and the specification are given in section 5, two slightly different versions of a controller are described in section 6. In section 7 it is proven formally, that the system consisting of controller and plant is actually a refinement of the specification. These proofs are rather technical and can be skipped on first reading. The paper concludes with some remarks and indications on further work to be done.

2 Descriptions of reactive systems

2.1 General considerations

A reactive system can not be seen in isolation from its environment: It is reacting on stimuli of its environment. An abstract view of a reactive system is depicted in figure 1, taken from [Ostroff, 1989, page 4]. In this figure the box labeled controller denotes the reactive system, the box labeled plant denotes the environment. Throughout this paper, the terms controller and plant will be used for the reactive system and its environment, respectively. The controller has to ensure, that the plant behaves in a certain desired way. To fulfill this task, the controller is able to sense the state of the plant – at least that part of the state which is relevant for this task. The controller is able to react to this information by sending appropriate stimuli to the plant. The task of the controller is complicated by the fact, that the plant is subject to disturbances. The controller is not able to sense these disturbances directly, only their effects on the plant can be 'seen' by it. One can say also, that the disturbances are outside the scope of the model of the plant, which is incorporated – explicitly or implicitly – in the controller.

The controller is seen here as an *open* system, which is interacting with the plant. On the

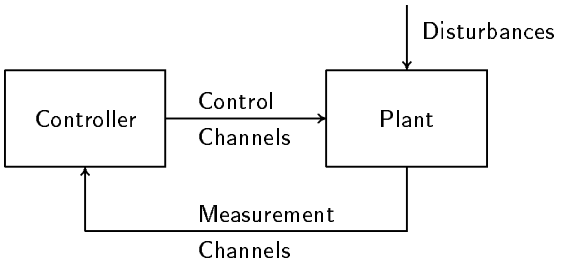


Figure 1: System consisting of plant and controller

contrary, the controller together with the plant – including the disturbances – can be viewed as a *closed* system without interaction with a further environment.

When developing a controller, at least three different descriptions have to be produced. At first, the plant has to be described. Thereby, a set of *possible* or *open-loop* behaviors is defined. At second, it must be possible to distinguish between desired and undesired behaviors of the plant. The *requirements* are the desired *closed-loop* behaviors, they are described by a *specification*. The closed-loop behaviors are behaviors of a system consisting of the controller as well as of the plant. Because they have to be related to the open-loop behaviors of the plant, they should be expressed in terms of the plant only. These descriptions and their relationships are suggested by [Ostroff, 1989] and more explicitly by Parnas et.al., see e.g. [Courtois and Parnas, 1993, van Schouwen *et al.*, 1993]. There, the open-loop behaviors of the plant are described by a relation NAT on the monitored and controlled variables, which are considered as functions over time. The requirements are captured by a relation REQ on the same variables. In [van Schouwen *et al.*, 1993] feasibility conditions of REQ with respect to NAT are formulated.

A third description is the *design*, which can be seen also as an abstract implementation, of a controller, developed to fulfill its task as described by the specification: Each behavior of a system consisting of this controller and the plant shall be a desired one. Note, that this relationship between the description of the controller and the specification has to be shown explicitly. In the terminology of [van Schouwen *et al.*, 1993], such a design is described by a relation SOF on input and output variables. These variables are related to the monitored and controlled variables, respectively. The relationships are described by two relations IN and OUT. Often it is possible to use observable events of the environment, which are already described in the description of the plant, in the design of the controller. Then the monitored variables and input variables can be identified and the relation IN can be neglected. The controlled and output variables and the relation OUT can be treated similarly.

After having identified different descriptions, it comes to the question, which formalism can be used conveniently in which description. Although there are a lot of formalisms in the literature to describe specifications and designs, there are few to describe the plant. Often, only the set of observable events is enumerated. It is also possible to describe the plant by the same means as a design. In this case, the plant is viewed as one component of a larger (closed) system, the controller as another one. This approach is e.g. advocated by Zave in her work on the language PAISLey, see [Zave and Yeh, 1981, Zave, 1991]. There the environment and the controller are described by processes. Although Parnas et.al. [Courtois and Parnas, 1993] advocate the description of the environment by the relation NAT, few is said about concrete formalisms besides their proposal to use a tabular notation. More attention is given to descriptions of the plant in the development of hybrid systems, see e.g. [Grossmann *et al.*, 1993] and [Antsaklis *et al.*, 1995]. There, the physical behavior of the plant together with discrete changes can be described.

For the descriptions of the requirements and a designed controller *single* and *dual language* approaches can be distinguished. In the single language approaches one formalism is used in both descriptions. Typical examples of this approach are

- Process algebras, where descriptions of processes are compared by bisimulation, e.g. CCS[2] [Milner, 1989].

[2] CCS can also be used with specifications based on a modal logic, following a dual language approach.

- Automata based techniques, where the plant and the controller are described by processes generating languages, and the specification consists of automata accepting languages, see e.g. [Kurshan, 1994].
- TLA [Lamport, 1994b], where the required behaviors as well as the set of possible behaviors of plant and controller are described using a temporal logic.

In dual language approaches the requirements are typically stated as properties in some kind of a logic. Designs are written down using formalisms, in which they can be expressed as models of the logic used for describing the requirements. Examples of this approach are

- ESM/RTTL [Ostroff, 1989], which combines state machines and a real-time temporal logic.
- Linear-time temporal logic [Manna and Pnueli, 1992, Manna and Pnueli, 1993] together with languages based on fair transition systems.
- Combinations of process algebra and trace logic, see e.g. [Olderog, 1991] or CSP, see [Hoare, 1985].

The use of these approaches is related to different specification styles. In [Pnueli, 1992] a *system specification style* and a *requirements specification style* are distinguished. These can be classified also as *model oriented* and *property oriented* styles, respectively. As can be seen from the examples above, in the dual language approaches, the specifications are described typically using some kind of logic. A specification can be considered as consisting of a list of properties. On the contrary, single language approaches are not necessarily model oriented. On the one hand, it is clear, that giving a process term of a process algebra is a model oriented specification. On the other hand, the automata of [Kurshan, 1994] are used to describe properties and hence are used in a property oriented manner.

An advantage of property oriented specifications is, that the requirements can be described explicitly. Disadvantages of this approach are, that the formulae describing the properties are often difficult to understand. Also it is a nontrivial task to show, that there is at least one behavior satisfying a specification. On the other hand it is often possible to visualize the models and thus aid understanding and validating them. A disadvantage of using models is that properties have to be encoded in them, they are given implicitly only.

2.2 A combination of specification styles

In this paper, a combination of these specification styles is proposed to get an expressive language for descriptions of reactive systems. Therefore a state-based language is embedded into a temporal logic. This combined language can be used for each of the three descriptions mentioned above. In each of these descriptions both parts of the combined language can be used, hence it will be possible to express different properties in their most convenient way. Hence, formulations shall both be precise as well as comprehensible. Because the different descriptions are given in the same combined language, it will be easily possible to relate them formally, e.g. by showing that a system consisting of a controller and the plant is a refinement of the specification. On the other hand, it will not be possible to avoid disadvantages of the single formalisms. E.g. descriptions might be inconsistent, i.e. there is no behavior satisfying it, and it is difficult to show, that a description is consistent actually.

The plant can be described by a set of states and transitions between them. Furthermore, it is possible to describe important properties of the plant explicitly by temporal formulae. E.g. it is possible to describe different modes of the plant and the transitions between them using the state-based part, whereas safety and liveness properties of the plant can be expressed by temporal formulae. Thus, the state-based part defines the signature necessary to state the requirements and assertions about the plant in the part based upon temporal logic.

The requirements can be stated by temporal formulae as it is typical for the property oriented style. In addition, if a specific sequence of actions or a protocol is part of the requirements, these can be expressed using the state-based part by writing down a corresponding model.

The design of a controller can be expressed by a purely state-based description. But it is also possible to refine successively the temporal formulae. In each refinement step some properties explicitly expressed by temporal formulae become implicit properties of a state-based model. A similar approach is taken in [Henzinger *et al.*, 1993] and [Olderog, 1991].

It is expected, that the description of the plant will still be biased towards a model oriented style, the requirements will be biased towards a property oriented style, whereas the design of a controller can be described by a model without further explicitly expressed properties.

The state-based language used in this paper is the formalism of statecharts [Harel, 1987]. Due to its hierarchical and orthogonal organization of states it can be used to give rather compact descriptions of large systems. In addition, it embodies formalisms to structure a system according to its functionalities and to express architectural aspects, i.e. activity and module charts, respectively[3].

To combine the languages, a statechart is viewed as a generator of linear sequences of states, whereby a state in such a sequence is corresponding to a configuration of the statechart. Given a statechart, it is shown then, how a temporal formula can be constructed, which has as its models the behaviors generated by the statechart. Because linear sequences are considered here, a linear time temporal logic can be used here. Therefore, TLA [Lamport, 1994b] has been chosen. It has been prefered to LTL [Manna and Pnueli, 1992] because the central notion of TLA, i.e. *actions*, resembles single steps of a statechart very closely. Furthermore, the concept of stuttering steps should make it easier to introduce a notion of refinement in the combined formalism.

Describing the sequences which are generated by a statechart by a temporal formula in TLA is done similar to the definition of the temporal semantics of fair transitions systems in [Manna and Pnueli, 1992] and the definition of a semantics of statecharts given by Day et.al. [Day, 1993, Day and Joyce, 1993]. A similar construction relating diagrams and TLA is given in [Lamport, 1995], there *predicate-action diagrams* are introduced and their semantics is defined by a translation to TLA formulae. These diagrams are used to illustrate specifications and proofs and to provide supplementary views of a system. Hence, these diagrams have a different role as the statecharts used in this paper.

[3]Because the example used in this paper is a small one, these formalisms have not been used here. Furthermore, alternatives to combine functional units have to be taken into account, see e.g. the work of Leveson et.al. on RSML [Leveson *et al.*, 1994].

3 Introduction to TLA

To make this report more selfcontained, a short introduction to TLA is given in this section. This is essentially along the lines of [Lamport, 1995], adapted to the needs of this paper. A comprehensive exposition of TLA can be found in [Lamport, 1994b]. The syntax of TLA formulae and the proofrules of TLA are summarized in the appendix.

TLA (Temporal Logic of Actions) is a temporal logic describing sets of linear behaviors. A *behavior* is an infinite sequence of states, and a *state* is an assignment of values to variables. It is assumed, that there is an infinite set of variables and a class of semantic values. A variable can be either a *flexible* or a *rigid* one. A *flexible variable* corresponds to variables in a programming language, different states may assign a different value to it. A *rigid variable* can be viewed as a constant, each state of a behavior assigns the same value to it. In principle, the variables are not restricted to be first order ones, but in this paper all flexible variables are first order ones. Higher order ones, i.e. functions and predicates, are rigid variables.

A *state function* is a non-boolean expression built from (first order) variables, constants, and constant operators, e.g. $x + 1$ is a state function. Semantically, a state function assigns a value to a state. For example, $x + 1$ assigns to a state s one plus the value that s assigns to x. A *state predicate* is a boolean expression built as above, its meaning is a mapping of states to boolean values.

A TLA *formula* is built up from state functions using the usual boolean operators, the temporal operators $'$ and $\Box$, and the hiding operators $\exists$ and $\mathbf{\exists}$. The following description is illustrated by the formula defined in figure 2. An *action* is a boolean expression containing primed and unprimed variables. Validity of an action is determined for a pair of states, thereby the primed variables are refering to the second state. If f is a state function or predicate, then f' denotes the expression obtained by priming the variables of f. As an example, action M_1 is true for the pair of states $\langle s, t\rangle$, iff the value that t assigns to x equals one plus the value assigned by s, and the values assigned to y are the same ones. A pair of states satisfying an action A is called an A step.

For an action A and a state function v, $[A]_v$ is defined as $A \vee (v = v')$. A $[A]_v$ step is either an A step, or a step which leaves the value of v unchanged, in the second case it is called a *stuttering* step. In the example, a $[M_1]_{\langle x,y\rangle}$ step is one that increments x and leaves y unchanged, or one which leaves the tuple $\langle x, y\rangle$ unchanged. $\langle A\rangle_v$ is defined as $A \wedge (v \neq v')$, so a $\langle M_1\rangle_{\langle x,y\rangle}$ step is one which changes x or y, which implies, that it increments x by one and leaves y unchanged.

Semantically, a TLA formula is true or false of a behavior. A predicate is true of a behavior iff it is true of the first state of it. An action is true of a behavior iff it is true of the first pair

---

$$
\begin{array}{lcl}
Init_\Phi & \triangleq & (x = 0) \wedge (y = 0) \\
M_1 & \triangleq & (x' = x + 1) \wedge (y' = y) \\
M_2 & \triangleq & (y' = y + 1) \wedge (x' = x) \\
M & \triangleq & M_1 \vee M_2 \\
\Phi & \triangleq & Init_\Phi \wedge \Box[M]_{\langle x,y\rangle} \wedge WF_{\langle x,y\rangle}(M_1) \wedge WF_{\langle x,y\rangle}(M_2)
\end{array}
$$

Figure 2: The TLA formula Φ describing a program that repeatedly increments x and y.

of states. For a formula F, the semantics of $\Box F$ is as usual: F is always true. E.g. $\Box[M]_{\langle x,y\rangle}$ is true of a behavior iff each step (pair of successive states) is a $[M]_{\langle x,y\rangle}$ step. Remember, that a behavior is a sequence of states, but not the sequence of actions taken to obtain a sequence of states.

A formula $Init \wedge \Box[N]_v$, where *Init* is a state predicate and N is an action, is a *safety property* according to [Alpern and Schneider, 1985]. It describes the possible initial states of a behavior – *Init* – and which steps are allowed – $[N]_v$. But it does not require anything to happen, it is satisfied by a behavior, which satisfies *Init* and each step of the behavior is a stuttering one.

Liveness properties express that something must happen. These are described by *fairness properties* in TLA. Then it is guaranteed, that no additional safety properties are implied by the liveness properties. In this case, a specification is called *machine closed*, see [Abadi and Lamport, 1994] for the complete definition of this term. Fairness operators are used to express that specific actions have to be taken. An action A is *enabled* in a state s iff there exists a state t, such that $\langle s, t\rangle$ is an A step. The *weak* fairness formula $WF_v(A)$ asserts of a behavior, that there are infinitely many $\langle A\rangle_v$ steps or there are infinitely many states in which $\langle A\rangle_v$ is not enabled. In other words, $\langle A\rangle_v$ cannot be enabled continuously without being taken. The strong fairness formula $SF_v(A)$ asserts that infinitely many $\langle A\rangle_v$ steps are taken or there are only finitely many states in which $\langle A\rangle_v$ is enabled. In other words, if $\langle A\rangle_v$ is enabled infinitely often, then infinitely many $\langle A\rangle_v$ steps occur.

The standard form of a TLA specification is $Init \wedge \Box[N]_v \wedge L$, where *Init* is a predicate, N is an action, v is a state function, and L is a conjunction of fairness conditions. This formula asserts of a behavior that

1. *Init* is true for the initial state,
2. every step of the behavior is an N step or leaves v unchanged, and
3. L holds.

Formula Φ of figure 2 is in this form, asserting that

1. initially x and y both equal zero,
2. every step either increments x by one and leaves y unchanged, increments y by one and leaves x unchanged, or leaves both x and y unchanged, and
3. the fairness condition $WF_{\langle x,y\rangle}(M_1) \wedge WF_{\langle x,y\rangle}(M_2)$ holds.

Formula $WF_{\langle x,y\rangle}(M_1)$ asserts that there are infinitely many $\langle M_1\rangle_{\langle x,y\rangle}$ steps or $\langle M_1\rangle_{\langle x,y\rangle}$ is infinitely often not enabled. Since 1) and 2) imply that x is always a natural number[4], $\langle M_1\rangle_{\langle x,y\rangle}$ is always enabled. Hence, $WF_{\langle x,y\rangle}(M_1)$ implies that there are infinitely many $\langle M_1\rangle_{\langle x,y\rangle}$ steps, so x is incremented infinitely often. Similarly, $WF_{\langle x,y\rangle}(M_2)$ implies, that y is incremented infinitely often. Hence, Φ is true of a behavior iff

1. x and y are initially zero,

[4]Note that TLA is an untyped logic.

2. every step increments either x or y by one and leaves the other unchanged or else leaves both x and y unchanged, and

3. both x and y are incremented infinitely often.

The operators $\exists$ and $\mathbf{\exists}$ are used to *hide* rigid and flexible variables, respectively. In the example, Φ asserts that x and y are incremented repeatedly, $\mathbf{\exists}\, y : \Phi$ expresses only, that x is incremented repeatedly. The formula $\mathbf{\exists}\, y : F$ is satisfied by a behavior iff there is some sequence of values which can be assigned to y which gives a behavior satisfying F. The precise definition is given in [Lamport, 1994b, page 904]. The operator $\exists$ can be defined in terms of $\mathbf{\exists}$, by stating that the hidden variable will not change its value.

The symbols $\exists$ and $\mathbf{\exists}$ are used already in TLA for quantification over variables. To describe the set of behaviors of a statechart, sets of states, events, etc. are used. These sets are flexible variables. $\bigvee$ and $\bigwedge$ will denote existential and universal quantification, respectively, over these sets. Because these sets are finite and fixed for a statechart, these symbols can also be understood as abbreviations for finite disjunction and finite conjunction, respectively. To shorten the formulae, $\bigwedge\limits_{s \in S_1} p(s)$ will be used as an abbreviation for $\bigwedge\limits_{s \in S} (s \in S_1 \Rightarrow p(s))$ and $\bigvee\limits_{s \in S_1} p(s)$ for $\bigvee\limits_{s \in S} (s \in S_1 \wedge p(s))$, when $S_1 \subseteq S$.

Although TLA is an untyped language, each definition of predicates and functions in the sequel will be preceded by a corresponding signature to help understanding its meaning. E.g. $p : S$ will define a predicate p on the elements of the set S.

Because the formulae describing programs tend to be large, the syntactical conventions of Lamport [Lamport, 1994a] are used to enhance readability of the formulae. Indentation and bulleting with logical operators are used to eliminate as much parentheses as possible. As an example, the formula $(a \vee b) \wedge (c \Rightarrow d)$ will be written as

$$
\begin{array}{l}
\wedge \vee a \\
\quad \vee b \\
\wedge c \\
\quad \Rightarrow d
\end{array}
$$

4 A subset of statecharts

A statechart can be considered as generating linear sequences of states. Each sequence is corresponding to one possible behavior of the statechart and each state in such a sequence consists of the actual configuration of the statechart and the events currently existing. Transitions between states of the statechart are used to describe which configuration is the next one. Which transitions can be taken actually in a step is depending on the current set of existing events. Besides changing the configuration, a new set of existing events is determined by a step. Also, it is possible to extend the set of existing events, thereby leaving the current configuration unchanged. Because there are only finitely many possible events, the set of events cannot be extended ad infinitum. A sequence of states generated by a statechart begins with the initial configuration, assuming the events necessary to enter it, and the set of events generated when entering this configuration. It is possible, that there exists no initial configuration, in this case the statechart corresponds to one behavior of zero length. Also,

when a state is reached from which no further step can be taken, then this behavior of the statechart is a finite sequence of states. Note, that in a behavior only the sequence of states is captured, but it does not contain information about the transitions actually taken. Hence it is not possible to distinguish two steps of the statechart changing the configuration in the same way and generating the same events.

In the following, the behaviors of a statechart are characterized by TLA formulae. At first, static properties of a statechart are described. This includes the hierarchical organization of states and properties of single transitions. Thereby syntactically correct statecharts are described. A configuration of the statechart, i.e. a set of active states, can be viewed as a state of a finite automaton.

The single transitions of the statechart are combined to so called compound ones, describing the change of control for an independent part of a configuration. Such a part is determined by the lowest common ancestor of the source and target states of the compound transition.

Several compound transitions can be combined to describe the transition from the current configuration to the next one. Whether a compound transition can be taken actually depends also on the existence of events. The sets of compound transitions, which can be taken together, correspond to the transitions between the states – the configurations of the statechart – of the finite automaton mentioned above.

At last, it is described by TLA actions, what it means to take such a set of compound transitions, i.e. how does the configuration change and which set of events is generated. These actions and a fairness property on the action corresponding to a step of the statechart are the main parts of the TLA formula describing the semantics of the statechart. Note, that not a new semantics of a statechart is defined here. The main purpose of this section is to give the semantics in such a way, that it is possible to combine a statechart with temporal formulae in a clean way.

In this paper the semantics of statecharts is not defined for the full formalism, only a subset[5], including the hierarchical and orthogonal organization of states, is considered. Triggering events, enabling conditions and actions are restricted: Only simple events and boolean combinations of them are used to trigger transitions, and as actions only simple events can be generated. As enabling conditions only conditions of the form in(state) and boolean combinations of them will be used. Neither data, time and history connectors are considered here, nor are activity or module charts.

At the end of the section it is considered whether the temporal semantics captures the behaviors of the statechart adequately. The temporal semantics as defined here is compared with semantics of statecharts known from literature.

4.1 Static aspects of statecharts

A statechart can be viewed as consisting of sets of states and transitions, respectively. In this section properties of these sets are described, thereby characterizing syntactically correct or well formed statecharts. These properties are the hierarchical organization of states and properties of single transitions with respect to possible source and target states. Then, it is described, how single transitions can be composed to compound transitions, which are used in turn to describe the possible steps of a statechart.

[5]The choice of this subset is motivated mainly by the subset of statecharts used in the example in section 5.

4.1.1 The hierarchical organization of states

A statechart consists of a finite, non-empty set of states S. These states describe possible locations of control. A state is called *active* if control is in it. In contrast to a finite automaton the states of a statechart may be organized hierarchically. Thus it is possible to structure the set of states and to have views of a statechart on different levels of abstractness. Each state is of one of three possible types. *basic*-states are at the bottom of the hierarchy. The direct descendants of an *or*-state resemble a finite automaton: If an *or*-state is active, then exactly one of its direct descendants is active, too. *and*-states are used to express some kind of parallelism or concurrency: If an *and*-state is active, then all its direct descendants are active simultaneously. These operate concurrently, but in a synchronized way. A *configuration* is a set of active states satisfying the conditions mentioned above with respect to the type of the states[6]. It is determined uniquely by the *basic*-states in it.

In the statechart in figure 3 the states A, D, and E are *or*-states, C is an *and*-state and B, F, G, H, and I are *basic*-states. The state C can be recognized as an *and*-state because of the dashed line separating its direct substates D and E and because the name of the state is written in an extra box outside the state boundary. One possible configuration[7] is $\{\mathsf{A}, \mathsf{C}, \mathsf{D}, \mathsf{G}, \mathsf{E}, \mathsf{H}\}$, which is determined uniquely by the basic states G and H.

The hierarchical relation between states is captured by the predicate *substate*. We define corresponding predicates for the transitive as well as for the transitive and reflexive closure of this relation.

$$
\begin{aligned}
&substate : S \times S) \\
&substate(s_1, s_2) \quad \triangleq \quad s_2 \text{ is a direct substate of } s_1, \\
&\qquad\qquad\qquad\quad \text{this depends on the actual statechart.}^{8}
\end{aligned}
$$

[6] It is defined formally later on, that taking transitions must lead to a correct configuration (3).
[7] Note, that this configuration is not reachable, nevertheless it is a correct one.
[8] Comments in a formula are simply written in roman font in the formula itself.

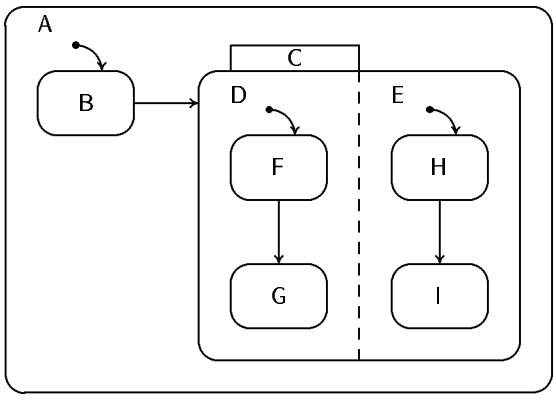


Figure 3: Hierarchical states

$$substate^+ : S \times S$$
$$substate^+(s_1, s_2) \triangleq \begin{array}{l} s_2 \text{ is a proper substate of } s_1 \\ \vee\ substate(s_1, s_2) \\ \vee\ \bigvee_{s_3 \in S} \wedge\ substate^+(s_1, s_3) \\ \qquad \wedge\ substate^+(s_3, s_2) \end{array}$$

$$substate^* : S \times S$$
$$substate^*(s_1, s_2) \triangleq \begin{array}{l} \vee\ s_1 = s_2 \\ \vee\ substate^+(s_1, s_2) \end{array}$$

The predicates *and*, *or*, and *basic* are used to check the type of the states.

$$basic : S$$
$$and : S$$
$$or : S$$

The predicate *correct.type* expresses, that a state is of exactly one of the three types. Furthermore, *basic*-states do not have substates and *and*-states cannot have other *and*-states as direct descendants[9].

$$correct.type : S$$
$$correct.type(s) \triangleq \begin{array}{l} \wedge \text{ A state without a descendant is a } basic\text{−state:} \\ \quad \neg \bigvee_{s_1 \in S} substate(s, s_1) \\ \quad \Rightarrow basic(s) \wedge \neg and(s) \wedge \neg or(s) \\ \wedge \text{ A state with descendants is either an } and\text{− or an } or\text{−state:} \\ \quad \bigvee_{s_1 \in S} substate(s, s_1) \\ \quad \Rightarrow \wedge\ \neg basic(s) \\ \qquad \wedge\ and(s)\ xor\ or(s) \\ \wedge \text{ An } and\text{−state does not have an } and\text{−state as a direct descendant:} \\ \quad and(s) \Rightarrow \bigwedge_{s_1 \in S} substate(s, s_1) \Rightarrow \neg and(s_1) \end{array}$$

The states of a statechart form a tree, where the root of the tree must be an *or*-state. This is the state A in the statechart in figure 3. The function *root* gives this root of a given statechart.

[9] In some of the statecharts in this paper this restriction will be violated. This does not cause any problems, because it is always possible to separate the *and*-states by an intermediate *or*-state.

$tree$: [10]

$$tree \triangleq \begin{array}{l} \wedge \text{ no state is a descendant of itself:} \\ \quad \bigwedge_{s \in s} \neg substate^{+}(s, s) \\ \wedge \text{ Each state is descendant of at most one state:} \\ \quad \bigwedge_{s_1, s_2, s_3 \in S} \begin{array}{l} \wedge\ substate(s_1, s_3) \\ \wedge\ substate(s_2, s_3) \\ \Rightarrow\ s_1 = s_2 \end{array} \\ \wedge \text{ Each two states have a common ancestor:} \\ \quad \bigwedge_{s_1, s_2 \in S} \bigvee_{s_3 \in S} \begin{array}{l} \wedge\ substate^{*}(s_3, s_1) \\ \wedge\ substate^{*}(s_3, s_2) \end{array} \\ \wedge \text{ All states are descendants of one } or\text{-state:} \\ \quad \bigvee_{s_1 \in S} \wedge \bigwedge_{s_2 \in S} substate^{*}(s_1, s_2) \\ \qquad \wedge\ or(s_1) \end{array}$$

$root$: $\rightarrow S$

$$s = root \triangleq \bigwedge_{s_1 \in S} substate^{*}(s, s_1)$$

These properties of a given statechart with set of states S are summarized by:

$hierarchy$:

$$hierarchy \triangleq \begin{array}{l} \wedge\ S \neq \emptyset \\ \wedge \bigwedge_{s \in S} correct.type(s) \\ \wedge\ tree \end{array}$$

4.1.2 Single transitions

Transitions describe how to reach one configuration from another one. They are build up from single transitions, relating source and target states. Besides the states, *connectors* can be used as source or target of transitions. Using these connectors, several single transitions can be combined to a *compound* transition. E.g. in the statechart shown in figure 4 there are three single transitions and two compound ones: One leaving state A and entering state B and one leaving A and entering C. Taking a transition will be considered as an atomic step. In the graphical language statecharts there are three different types of connectors, viz *condition*, *switch*, and *junction* connectors. They indicate whether the selection, which of the compound transitions has to be taken, is based on the triggering events of the transition (switch connector) or on the enabling conditions of the transition (condition connector). Junction connectors should be used in the remaining cases. Because there is no semantic difference between these types of connectors it is sufficient to define just one set of connectors, denoted C. This set is disjoint to the set of states S, S_C denotes the union of S and C. The connectors are not related hierarchically neither to the states nor to themselves. Note,

[10] The empty signature is indicating that this predicate is one which expresses properties of the sets constituting the statechart. Here this is the set of states S.

that control can only be in states, but not in connectors. In this section only the existence of connectors is important, in the next one it is shown, how single transitions can be composed to compound transitions.

Already a single transition can have several source and target states (or connectors). These transitions are expressed graphically by *fork* and *joint* constructs, respectively. As an example see the statechart in figure 5.

The set of transitions of a given statechart is denoted by T and is assumed to be finite. The functions *state* and *target* are used to access the source and the target states (including connectors) of a transition. Naturally, each transition has non-empty sets of source and target states.

$$source : T \to 2^{S_C}$$
$$source(t) \triangleq \text{The set of source states (and connectors) of } t.$$

$$target : T \to 2^{S_C}$$
$$target(t) \triangleq \text{The set of target states (and connectors) of } t.$$

$$source.target : T$$
$$source.target(t) \triangleq \wedge\ source(t) \neq \emptyset$$
$$\wedge\ target(t) \neq \emptyset$$

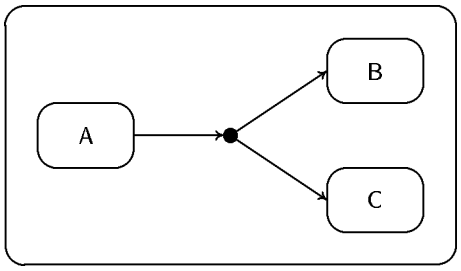


Figure 4: Compound transitions

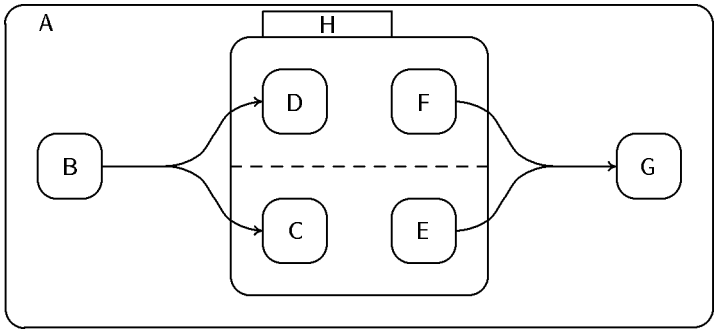


Figure 5: Fork and Joint constructs

The possible sets of source and target states of one transition are restricted by the hierarchical relation and the type of the states: It must be possible, that all source states of a transition can be active simultaneously. E.g. two direct descendants of an *or*-state cannot be active simultaneously. Furthermore, the source states must be independent in the sense, that if one of the source states is active, then this does not imply that another one is active too, i.e. the states must not be related in the hierarchy of states. If two states satisfy these properties, they will be called *orthogonal*. See figure 6 for a statechart with two non-orthogonal transitions. The source states of the left-hand transition are not independent. For the right hand transition it is not possible, that both source states of it are active simultaneously, because these are direct descendants of the same *or*-state E.

There are similar restrictions for the target states of a transition, but not for connectors, because these cannot be active. Two states can be active simultaneously if their lowest common ancestor with respect to the hierarchical relation of states is not an *or*-state. Two states are independent of each other if one is not a substate of the other one.

$$
\begin{aligned}
&common.ancestor : S \times S \rightarrow 2^S \\
&s_3 \in common.ancestor(s_1, s_2) \triangleq \ \wedge\ substate^*(s_3, s_1) \\
&\qquad\qquad\qquad\qquad\qquad\quad \wedge\ substate^*(s_3, s_2)
\end{aligned}
$$

$$
\begin{aligned}
&lowest.common.ancestor : S \times S \rightarrow S \\
&s_3 = lowest.common.ancestor(s_1, s_2) \triangleq \ \wedge\ s_3 \in common.ancestor(s_1, s_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge \bigwedge_{s_4 \in S} s_4 \in common.ancestor(s_1, s_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow substate^*(s_4, s_3)
\end{aligned}
$$

$$
\begin{aligned}
&orthogonal : S \times S \\
&orthogonal(s_1, s_2) \triangleq \ \wedge\ \neg or(lowest.common.ancestor(s_1, s_2)) \\
&\qquad\qquad\qquad\quad \wedge\ \neg substate^*(s_1, s_2) \\
&\qquad\qquad\qquad\quad \wedge\ \neg substate^*(s_2, s_1)
\end{aligned}
$$

A transition is called *disjoint*, if all its source and target states are orthogonal.

$$
\begin{aligned}
&disjoint : T \\
&disjoint(t) \triangleq \bigwedge_{s_1, s_2 \in S} s_1, s_2 \in source(t) \cup target(t) \\
&\qquad\qquad\qquad\qquad\quad \Rightarrow \vee\ orthogonal(s_1, s_2) \\
&\qquad\qquad\qquad\qquad\qquad\quad \vee\ s_1 = s_2
\end{aligned}
$$

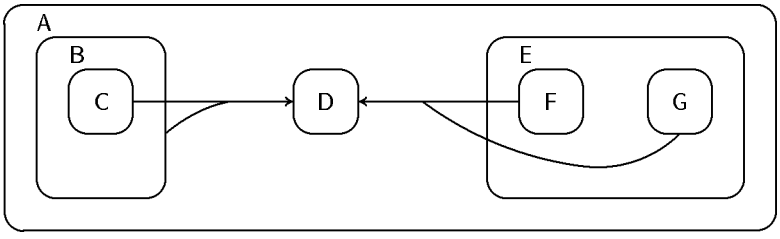


Figure 6: Non-orthogonal transitions

When an *or*-state becomes active it must be specified, which of its descendants will become active, too. This is done by using *default transitions*. E.g. the transition leading to state D in the statechart in figure 3 is such a default transition. When entering the *or*-state D, then the state F is entered too, but not the state G. The set of default transitions is denoted by D, with $D \subseteq T$. We define the (single) source state of a default transition to be the corresponding *or*-state. For each *or*-state there has to be exactly one default transition. If the set of targets of a default transition contains states, then these states has to be proper descendants, but not necessarily direct ones, of the source state.

$$
\begin{aligned}
&exactly.one.default : S \\
&exactly.one.default(s) \triangleq or(s) \Rightarrow \wedge \bigvee_{d \in D} source(d) = \{s\} \\
&\qquad \wedge \bigwedge_{d_1, d_2 \in D} \begin{array}{l} \wedge\ source(d_1) = \{s\} \\ \wedge\ source(d_2) = \{s\} \\ \Rightarrow d_1 = d_2 \end{array}
\end{aligned}
$$

$$
\begin{aligned}
&correct.default : D \\
&correct.default(d) \triangleq \wedge \bigvee_{s \in S} \begin{array}{l} \wedge\ or(s) \\ \wedge\ source(d) = \{s\} \end{array} \\
&\qquad \wedge \bigwedge_{s_1, s_2 \in S} \begin{array}{l} \wedge\ \{s_1\} = source(d) \\ \wedge\ s_2 \in target(d) \\ \Rightarrow substate^+(s_1, s_2) \end{array}
\end{aligned}
$$

In a correct statechart, each of the transitions of T is disjoint and has at least one source and one target state, and each of the default transitions is correct.

$$
\begin{aligned}
&correct.single.transition : \\
&correct.single.transition \triangleq \wedge \bigwedge_{t \in T} \begin{array}{l} \wedge\ disjoint(t) \\ \wedge\ source.target(t) \end{array} \\
&\qquad \wedge \bigwedge_{s \in S} exactly.one.default(s) \\
&\qquad \wedge \bigwedge_{d \in D} correct.default(d)
\end{aligned}
$$

A correct or *well formed* statechart is described by these constraints on transitions and by the constraints on the set of states.

$$
\begin{aligned}
&well.formed : \\
&well.formed \triangleq \wedge\ hierarchy \\
&\qquad \wedge\ correct.single.transition
\end{aligned} \tag{1}
$$

4.1.3 Compound transitions

Using single transitions to describe more complex, compound ones, can make the charts more readable. In this section it is defined how single transitions can be connected to form compound transitions. A compound transition should behave essentially like one (large) single

transition. In the following a compound transition is viewed as a set of single transitions, characterized by properties of this set.

Each compound transition consists of two parts. The first one relates source and target states by a set of single transitions. Thereby intermediate connectors may be used, but no default transitions. The second part extends the first one towards basic states, such that by taking the compound transition, a configuration will be reached.

As examples see the statecharts depicted in the figures 7[11] and 8, the second one being taken from [Harel, 1987, page 264]. In the statechart in figure 7 it is possible to take one of the transitions $\{t_1, t_2\}$ or $\{t_1, t_3\}$ when leaving the state B. These transitions do not have a second part, because by entering the states D and F, resp., exactly one of the substates of the *or*-state C becomes active already. A transition with target state A however will have as its second part the default transitions d_1 and d_2, which must be taken to ensure that the *or*-states A and C both have an active descendant. In figure 8 one of the following sets of transitions can be taken in one step: $\{t_1, t_2\}$, $\{t_3, d_2\}$, $\{t_4, d_1\}$. Note, that the set of

[11]The labels of the transitions in these figures have no meaning, they are used only for referencing the transitions in the text.

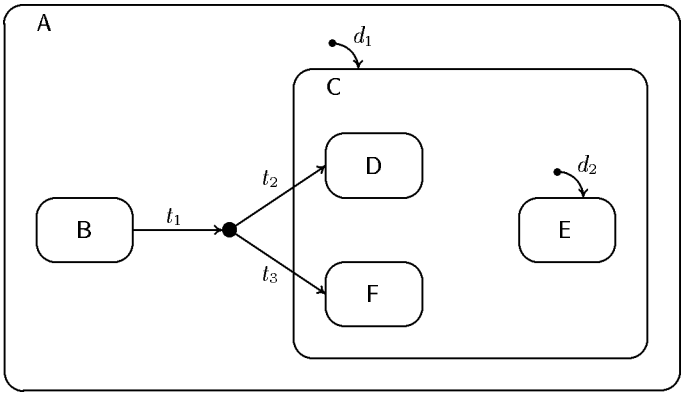


Figure 7: Compound transitions, first example

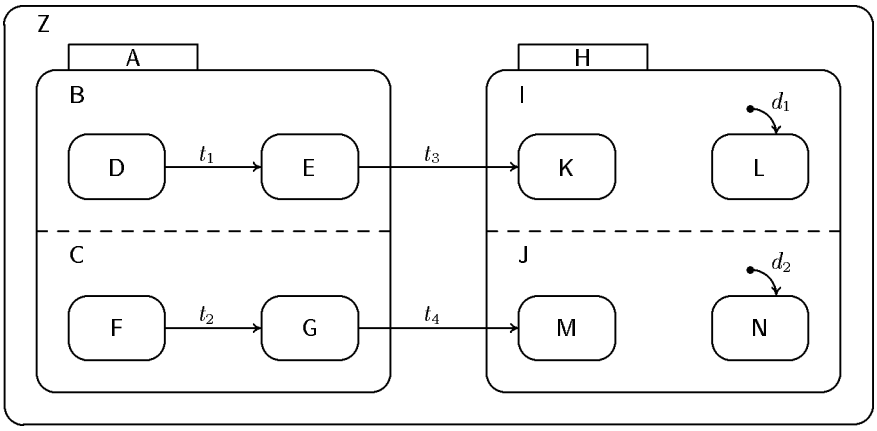


Figure 8: Compound transitions, second example

transitions $\{t_3, t_4\}$ will not be considered as one compound transition, because its first part can be split into two independent compound transitions, i.e. $\{t_3\}$ and $\{t_4\}$.

As can be imagined from these small examples, single transitions can be combined in rather complex ways to compound transitions. As a consequence, it is a nontrivial task to determine which single transitions form a compound one. With respect to practical usage and reviewability of descriptions, compound transitions should be combined from single ones in simple ways.

Compound transitions are defined by regarding the states and transitions as a directed graph. To define the first part of a compound transition *paths* of states and connectors corresponding to a set of transitions are defined. A path with respect to a set of transitions is a set of states and connectors which can be ordered linearly by starting from a state or connector and reaching another one by taking one transition. A path is an *extended* one, if it is not possible to add further states or connectors to it without losing the property of being a path.

$$
\begin{array}{l}
path : 2^{S_C} \times 2^T \\
path(S_1, T_1) \triangleq \quad \vee \text{ The empty set is a path:} \\
\qquad\qquad S_1 = \emptyset \\
\qquad \vee \text{ A singleton set is a path, if the element is a source or a target state:} \\
\qquad\qquad \bigvee_{s \in S_C} \bigvee_{t \in T_1} \wedge S_1 = \{s\} \\
\qquad\qquad\qquad\qquad \wedge \vee \ s \in source(t) \\
\qquad\qquad\qquad\qquad\quad \vee \ s \in target(t) \\
\qquad \vee \text{ There is a first state in the sequence and the other states are a path:} \\
\qquad\qquad \bigvee_{s_1, s_2 \in S_1} \bigvee_{t \in T_1} \wedge \ s_1 \in source(t) \\
\qquad\qquad\qquad\qquad \wedge \ s_2 \in target(t) \\
\qquad\qquad\qquad\qquad \wedge \ path(S_1 \setminus \{s_1\}, T_1)
\end{array}
$$

$$
\begin{array}{l}
extended.path : 2^{S_C} \times 2^T \\
extended.path(S_1, T_1) \triangleq \quad \wedge \ path(S_1, T_1) \\
\qquad\qquad\qquad\qquad \wedge \bigwedge_{s \in S_C} \ s \notin S_1 \\
\qquad\qquad\qquad\qquad\qquad \Rightarrow \neg path(S_1 \cup \{s\}, T_1)
\end{array}
$$

A critical point is the existence of cycles in paths. There is no useful semantics of taking a compound transition containing a cycle consisting of connectors only: this cycle could be taken over and over again in one step. But not all cycles can be excluded, because it is possible to have transitions with source states which are target states too. Hence it is required, that paths which can be made circular by the transitions must contain all states of the path in the cycle. All extended paths of a correct first part of a compound transition must have a source and a target state, connectors are not allowed in this place. If these states are different, one of them must not be reachable via the compound transition, implying, that is is a start state. Therefore, the cycles corresponding to extended paths are the harmless ones: Each state (or connector) of the path is in the cycle, and the cycle contains exactly one state. Cycles corresponding to non-extended paths will be excluded by requiring, that each path is a subset of an extended path with respect to the same set of transitions, see the definition of *complete.compound* (2), page 18.

$$
\begin{array}{l}
reaches : S_C \times S_C \times 2^T \\
reaches(s_1, s_2, T_1) \triangleq \lor \text{ Starting from } s_1 \text{ it is possible to reach } s_2 \\
\qquad \text{via the transitions in } T_1 \\
\qquad \bigvee_{t \in T_1} \land\ s_1 \in source(t) \\
\qquad\qquad \land \lor\ s_2 \in target(t) \\
\qquad\qquad\qquad \lor \bigvee_{s_3 \in S_C} \land\ s_3 \in target(t) \\
\qquad\qquad\qquad\qquad \land\ reaches(s_3, s_2, T_1) \\
\qquad \lor\ s_1 = s_2
\end{array}
$$

$$
\begin{array}{l}
correct.path : 2^{S_C} \times 2^T \\
correct.path(S_1, T_1) \triangleq \lor \text{ The path contains two states and is acyclic:} \\
\qquad \bigvee_{s_1, s_2 \in S} \land\ S_1 \cap S = \{s_1, s_2\} \\
\qquad\qquad \land \bigwedge_{s_3 \in S_1} \land\ reaches(s_1, s_3, T_1) \\
\qquad\qquad\qquad \land\ reaches(s_3, s_2, T_1) \\
\qquad\qquad \land \neg reaches(s_2, s_1, T_1) \\
\qquad \lor \text{ The path contains one state and is cyclic:} \\
\qquad \bigvee_{s \in S} \land\ S_1 \cap S = \{s\} \\
\qquad\qquad \land \bigwedge_{s_1 \in S_1} \land\ reaches(s, s_1, T_1) \\
\qquad\qquad\qquad \land\ reaches(s_1, s, T_1)
\end{array}
$$

Similar to a single transition, all the source resp. target states of the first part of a compound transition must be orthogonal, i.e. the compound transition is disjoint.

$$
\begin{array}{l}
disjoint.compound : 2^T \\
disjoint.compound(T_1) \triangleq \bigwedge_{t_1, t_2 \in T_1} \bigwedge_{s_1, s_2 \in S} \land\ s_1 \in source(t_1) \cup target(t_1) \\
\qquad\qquad\qquad\qquad\qquad\qquad \land\ s_2 \in source(t_2) \cup target(t_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad \Rightarrow \lor\ orthogonal(s_1, s_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \lor\ s_1 = s_2
\end{array}
$$

One compound transition should correspond to one part of a step, which is independent of others. Therefore it must not be possible to separate one compound into several different compounds. On the other hand, it should consist of as many single transitions as possible, under the restriction above and that it is a disjoint compound transition. A compound contains as much transitions as possible if each extended sequence corresponding to this compound is also a correct one: each path starts with a state, ends with a state and contains only intermediate connectors.

$$
\begin{array}{l}
complete.compound : 2^T \qquad (2) \\
complete.compound(T_1) \triangleq \land\ disjoint.compound(T_1) \\
\qquad \land \text{ Each path with respect to } T_1 \text{ is an extended one:} \\
\qquad\quad \bigwedge_{S_1 \in 2^{S_C}} path(S_1, T_1) \\
\qquad\qquad \Rightarrow \bigvee_{S_2 \in 2^{S_C}} \land\ S_2 \supseteq S_1 \\
\qquad\qquad\qquad\qquad \land\ extended.path(S_2, T_1) \\
\qquad \land \text{ Each extended path is a correct path,} \\
\qquad\quad \text{thus excluding cycles containing connectors only:} \\
\qquad\quad \bigwedge_{S_1 \in 2^{S_C}} extended.path(S_1, T_1) \\
\qquad\qquad \Rightarrow correct.path(S_1, T_1)
\end{array}
$$

$$
\begin{aligned}
&maximal.compound : 2^T \\
&maximal.compound(T_1) \triangleq\ \wedge\ \text{No default transitions can be used in the first part:} \\
&\qquad T_1 \cap D = \emptyset \\
&\qquad \wedge\ \text{The compound transition is a complete one:} \\
&\qquad\ complete.compound(T_1) \\
&\qquad \wedge\ \text{The compound transition does not contain other} \\
&\qquad\ \text{complete compound transitions:} \\
&\qquad\ \bigwedge_{T_2 \in 2^T} \wedge\ T_2 \neq \emptyset \\
&\qquad\qquad\qquad \wedge\ T_2 \subsetneq T_1 \\
&\qquad\qquad\qquad\ \Rightarrow \neg complete.compound(T_2)
\end{aligned}
$$

In the example of figure 7 the only maximal compound transitions are $\{t_1, t_2\}$ and $\{t_1, t_3\}$. A compound consisting of just one of t_1, t_2, or t_3 contains extended paths which are not correct, because either the source or the target is a connector. The compound transition $\{t_1, t_2, t_3\}$ is not maximal because it is not disjoint: its target states t_2 and t_3 are not orthogonal. In the statechart in figure 8 each of the singletons $\{t_1\}$, $\{t_2\}$, $\{t_3\}$, and $\{t_4\}$ is already a complete compound transition. Therefore it is not possible to extend them, especially $\{t_3, t_4\}$ is not a maximal compound transition.

The second part of a compound transition extends the first one towards *basic*-states. The typical situation is, that from an *or*-state there is a default transition to one of its direct descendants. But as more general cases are allowed using connectors and transitions with several targets, the properties are more complex to express.

The functions *sources* and *targets* are used to access the source and target states of compound transitions.

$$
\begin{aligned}
&targets : 2^T \to 2^S \\
&s \in targets(T_1) \triangleq \bigvee_{t \in T_1} s \in target(t)
\end{aligned}
$$

$$
\begin{aligned}
&sources : 2^T \to 2^S \\
&s \in sources(T_1) \triangleq \bigvee_{t \in T_1} s \in source(t)
\end{aligned}
$$

This second part is assembled from default transitions and transitions which have only connectors as source states. Paths corresponding to the second part must be *descending* in the hierarchy of states. That is, if a state can be reached from another one via transitions, then it must be a substate of it. Descending paths are extended paths required to start and end with states, not with connectors.

In the case that an *and*-state is the target of a default transition, the default transitions of its components do not start at the *and*-state itself, but at the *or*-states directly below it. To bridge this gap in the descending paths we extend the reachability relation such that a descendant of an *and*-state can be reached from the *and*-state itself.

$$
\begin{aligned}
&and.reaches : S_C \times S_C \times 2^T \\
&and.reaches(s_1, s_2, T_1) \triangleq \; \vee\, and(s_1) \Rightarrow \bigvee_{s_3 \in S} \wedge\, substate(s_1, s_3) \\
&\qquad\qquad \wedge \vee\, s_3 = s_2 \\
&\qquad\qquad\quad \vee\, and.reaches(s_3, s_2, T_1) \\
&\qquad \vee \neg and(s_1) \Rightarrow \bigvee_{t \in T_1} \wedge\, s_1 \in source(t) \\
&\qquad\qquad \wedge \vee\, s_2 \in target(t) \\
&\qquad\qquad\quad \vee \bigvee_{s_3 \in S_C} \wedge\, s_3 \in target(t) \\
&\qquad\qquad\qquad \wedge\, and.reaches(s_3, s_2, T_1)
\end{aligned}
$$

$$
\begin{aligned}
&descending.path : 2^{S_C} \times 2^T \\
&descending.path(S_1, T_1) \triangleq \; \wedge \text{ The states in } S_1 \text{ are descending:} \\
&\qquad \bigwedge_{s_1, s_2 \in S_1 \cap S} and.reaches(s_1, s_2, T_1) \Rightarrow substate^+(s_1, s_2) \\
&\qquad \wedge\, S_1 \text{ starts with a state and ends with a } basic\text{-state} \\
&\qquad \bigvee_{s_1, s_2 \in S_1 \cap S} \bigwedge_{s_3 \in S_1} \wedge\, basic(s_2) \\
&\qquad\qquad \wedge\, s_1 \neq s_3 \\
&\qquad\qquad\quad \Rightarrow and.reaches(s_1, s_3, T_1) \\
&\qquad\qquad \wedge\, s_2 \neq s_3 \\
&\qquad\qquad\quad \Rightarrow and.reaches(s_3, s_2, T_1) \\
&\qquad \wedge \text{ Each transition with a source state is a default one} \\
&\qquad \bigwedge_{t \in T_1} source(t) \cap S \neq \emptyset \Rightarrow t \in D
\end{aligned}
$$

Extending the first part of a compound transition towards a configuration implies, that if below an *and*-state there is a target state, there must be target states below all of its direct descendants. Similarly, there must be target states below exactly one of the descendants of an *or*-state if there is a target state below it.

$$
\begin{aligned}
&correct.target : S \times 2^T \qquad (3) \\
&correct.target(s, T_1) \triangleq \bigvee_{s_1 \in S} \wedge\, s_1 \in targets(T_1) \\
&\qquad \wedge\, substate^*(s, s_1) \\
&\qquad \Rightarrow \wedge\, and(s) \Rightarrow \bigwedge_{s_2 \in S} substate(s, s_2) \\
&\qquad\qquad \Rightarrow \text{ There is target below each descendant of } s\text{:} \\
&\qquad\qquad\quad \bigvee_{s_3 \in S} \wedge\, s_3 \in targets(T_1) \\
&\qquad\qquad\qquad \wedge\, substate^*(s_2, s_3) \\
&\qquad \wedge\, or(s) \Rightarrow \wedge \text{ There is a target below at least one descendant of } s\text{:} \\
&\qquad\qquad \bigvee_{s_2, s_3 \in S} \wedge\, substate(s, s_2) \\
&\qquad\qquad\qquad \wedge\, s_3 \in targets(T_1) \\
&\qquad\qquad\qquad \wedge\, substate^*(s_2, s_3) \\
&\qquad\qquad \wedge \text{ There is a target below at most one descendant of } s\text{:} \\
&\qquad\qquad\quad \bigwedge_{s_2, s_3, s_4, s_5 \in S} \wedge\, substate(s, s_2) \\
&\qquad\qquad\qquad\qquad \wedge\, substate(s, s_3) \\
&\qquad\qquad\qquad\qquad \wedge\, substate^*(s_2, s_4) \\
&\qquad\qquad\qquad\qquad \wedge\, substate^*(s_3, s_5) \\
&\qquad\qquad\qquad\qquad \wedge\, \{s_4, s_5\} \subseteq targets(T_1) \\
&\qquad\qquad\qquad\qquad \Rightarrow s_2 = s_3
\end{aligned}
$$

This property concerning target states is restricted by the part of the statechart which can be influenced by a compound transition. This part is characterized by the *scope* of the compound transition, i.e. the lowest common ancestor of all source and states of the transition which is an *or*-state.

$$
\begin{aligned}
&common.ancestors : 2^T \to 2^S \\
&s \in common.ancestors(T_1) \quad \triangleq \quad \bigwedge_{s_1 \in S} s_1 \in sources(T_1) \cup targets(T_1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow substate^*(s, s_1)
\end{aligned}
$$

$$
\begin{aligned}
&scope : 2^T \to S \\
&s = scope(T_1) \quad \triangleq \quad \wedge\ or(s) \\
&\qquad\qquad\qquad\qquad\quad \wedge\ s \in common.ancestors(T_1) \\
&\qquad\qquad\qquad\qquad\quad \wedge \bigwedge_{s_1 \in S} \wedge\ s_1 \in common.ancestors(T_1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ or(s_1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow substate^*(s_1, s)
\end{aligned}
$$

The second part of a compound transition must not consist of to many transitions. Some of them are already exhibited by the property of correct target states. But there may be paths starting from different states in the hierarchy, but leading to the same target state. These are excluded by requiring that for each two paths leading to the same target, the starting states of these paths must be equal too.

$$
\begin{aligned}
&same.start.path : 2^{S_C} \times 2^{S_C} \times 2^T \\
&same.start.path(S_1, S_2, T_1) \quad \triangleq \quad \bigvee_{s_1 \in S} s_1 \in targets(T_1) \cap S_1 \cap S_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \bigvee_{s_2 \in S} \wedge\ s_2 \in S_1 \cap S_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge \bigwedge_{s_3 \in S} s_3 \in S_1 \cup S_2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow and.reaches(s_2, s_3, T_1)
\end{aligned}
$$

To connect the two parts of a compound transition, it is required, that below each non-basic target state of the first part, there is a target state of the second part of the transition. If a target state of the first part is already a basic state, this state must not be a target state of the second part.

$$
\begin{aligned}
&connected.parts : 2^T \times 2^T \\
&connected.parts(T_1, T_2) \quad \triangleq \quad \bigwedge_{s_1 \in S} s_1 \in targets(T_1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow \wedge\ \neg basic(s_1) \ \Rightarrow \bigvee_{s_2 \in S} \wedge\ s_2 \in targets(T_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ substate^+(s_1, s_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ basic(s_1) \ \Rightarrow \ s_1 \notin targets(T_2)
\end{aligned}
$$

Now it is possible to collect all properties of a compound transition T_1. It consists of two disjoint sets of transitions T_2 and T_3. T_2 is a maximal compound transition, which also defines the scope of T_1. T_2 is extended by T_3 to reach a configuration. Therefore each state below (and including) the scope of T_1 must be correct with respect to targets. Below each (non-*basic*) target state of the first part there must be a target state of the overall

compound. The transitions in the second part are restricted to form descending paths with some restrictions on the beginning of these paths.

Up to now it has been assumed tacitly, that the first part of a compound transition contains at least one single transition. But there is one special case where this assumption is not valid: When starting the execution of a statechart there will be a compound with an empty first part and a second part containing the default transition of the root state.

$$
\begin{aligned}
&one.compound : 2^T \qquad (4)\\
&one.compound(T_1) \triangleq \bigvee_{T_2,T_3\in 2^T} \wedge\ T_1 = T_2 \cup T_3 \\
&\qquad \wedge\ \emptyset = T_2 \cap T_3 \\
&\qquad \wedge\ T_2 \neq \emptyset \Rightarrow \wedge\ maximal.compound(T_2) \\
&\qquad\qquad\qquad \wedge\ scope(T_1) = scope(T_2) \\
&\qquad \wedge\ T_2 = \emptyset \Rightarrow \bigvee_{t\in D} \wedge\ t \in T_3 \\
&\qquad\qquad\qquad \wedge\ \{root\} = source(t) \\
&\qquad \wedge\ connected.parts(T_2, T_3) \\
&\qquad \wedge \bigwedge_{s\in S} substate^*(scope(T_1), s) \Rightarrow correct.target(s, T_1) \\
&\qquad \wedge \bigvee_{S_1\in 2^S C} extended.path(S_1, T_3) \Rightarrow descending.path(S_1, T_3) \\
&\qquad \wedge \bigwedge_{S_1,S_2\in 2^S C} \wedge\ descending.path(S_1, T_3) \\
&\qquad\qquad\qquad \wedge\ descending.path(S_2, T_3) \\
&\qquad\qquad\qquad \Rightarrow same.start.path(S_1, S_2, T_3)
\end{aligned}
$$

In the statechart shown in figure 7 the maximal compound transition $\{t_1, t_2\}$ cannot be extended by any of the default transitions. The target state of t_2 is the *basic* target state D, adding the default transition d_2 would introduce the *basic* target state E, which is not orthogonal to D. Therefore the predicate $correct.targets(\mathsf{C}, \{t_1, t_2, d_2\})$ is not satisfied. But without d_2 the default transition d_1 cannot be extended to a *basic* target state. Therefore neither d_1 nor d_2 can be added to $\{t_1, t_2\}$. But $\{t_1, t_2\}$ satisfies the predicate *one.compound*. The case of the compound transition $\{t_1, t_3\}$ is completely similar.

In figure 8 there are three different cases when the state A is active: The states {D,F}, {E,F}, and {E,G} can be active, respectively. The case {D,G} is symmetrical to the case {E,F}. In the case {D,F}, each of the singletons $\{t_1\}$ and $\{t_2\}$ is a maximal compound transition which cannot be extended further. In the case {E,F} there are the maximal compound transitions $\{t_2\}$ and $\{t_3\}$, again $\{t_2\}$ cannot be extended. But $\{t_3\}$ does not lead to a correct configuration. There is a *basic* target state in the substate I of the *and*-state H, but none in the substate J. Therefore the default transition d_2 must be added to $\{t_3\}$. It is not possible to add t_4 to $\{t_3\}$ because it is a transition with a source state, but not a default transition. The case {E,G} is similar to the last one, one gets the compound transitions $\{t_3, d_2\}$ and $\{t_4, d_1\}$. Which of these compound transitions can be taken simultaneously in one step is clarified in the next section.

4.2 Dynamic aspects of statecharts

The compound transitions defined in the previous section are possible candidates which can be taken in one step of a statechart. Whether a compound transition will be taken actually

depends on whether it is possible to take it in the current configuration of the statechart and how conflicts between conflicting compound transitions are resolved.

4.2.1 Executing compound transitions

A compound transition is *enabled* in a configuration of a statechart if all its source states are active. In addition, it depends on the existence of *events*, whether it can be taken actually. For a given statechart, there is a set E of simple events. In each state of a behavior of a statechart a subset of E will be existing. A transition can be taken only, if all the events associated with it are currently existing. Using a not operator, it can also be expressed, that certain events must not exist, but this has to be used with care, see the discussion below. Hence, a state in a behavior can be described by two flexible variables S_{active} and E_{exists} denoting the sets of active states and existing events, respectively. The predicate *in* is used to check whether a state is currently active.

$$S_{active} \subseteq S \tag{5}$$

$$E_{exists} \subseteq E \tag{6}$$

$$\begin{aligned} &in : S \\ &in(s) \triangleq s \in S_{active} \end{aligned} \tag{7}$$

To be more precise as above, a finite set of *event expressions* can be associated with a single transition. The function *event* is used to access the set of event expressions of a single transition. Event expressions are build up from simple events using the operators **and**, **or**, and **not**. The set of event expressions used in a statechart is denoted by E_{expr}.

$$event : T \rightarrow 2^{E_{expr}}$$

For each event expression e a corresponding predicate $check.event.\mathsf{e}$ is constructed to define its meaning[12]. When e is corresponding to a simple event e, then this predicate is defined as

$$check.event.\mathsf{e} \triangleq e \in E_{exists}$$

Let $\mathsf{e}_1, \mathsf{e}_2$ be event expressions and let e_3 be e_1 **and** e_2, e_4 be e_1 **or** e_2, and e_5 be **not** e_1, then

$$\begin{aligned} check.event.\mathsf{e}_3 &\triangleq \wedge\ check.event.\mathsf{e}_1 \\ &\quad\ \wedge\ check.event.\mathsf{e}_2 \\ check.event.\mathsf{e}_4 &\triangleq \vee\ check.event.\mathsf{e}_1 \\ &\quad\ \vee\ check.event.\mathsf{e}_2 \\ check.event.\mathsf{e}_5 &\triangleq \neg check.event.\mathsf{e}_1 \end{aligned}$$

[12] Thereby the use of second order logic, as by Day et.al. [Day, 1993, Day and Joyce, 1993], is avoided. The higher order part is hidden here in the construction of the predicates $check.event.\mathsf{e}$.

It can be argued, that negating an event is not intuitive, because an event can be considered as being instantaneous. Almost always the event will not be existing and almost always a transition with this negated event expression will be enabled. But in conjunction with another unnegated event, negation can make sense. Consider e.g. the expression e_1 **and not** e_2. Then it is only interesting whether e_2 exists, in those moments, in which e_1 is already existing. Note also, that negation destroys monotonicity. Without negation, additional events cannot disable transitions. Nevertheless, negation is useful to express priorities between transitions, see e.g. the event expressions in the statechart in figure 9. Let us assume, that the states D and G are active and the events e_1 and e_2 are existing. Then it is possible to take the transition from G to H as well as from that from G to I, whereas it is only possible to take the transition from D to F.

Event expressions can also be of the form e[c], where e is an event and c is a condition. The conditions are regarded as a subset of E_{expr}, denoted by C_{expr}. To check validity of conditions, predicates *check.condition.c* similar to *check.event.e* are used. These predicates are also defined recursively, the base case is restricted to conditions of the form in(s), where s is a state.

$$check.condition.\mathsf{in(s)} \triangleq in(s)^{13}$$

Boolean combinations of conditions are defined similarly to those of events. Now, from the event expression e[c] and its special case [c], the following predicates are constructed:

$$check.event.\mathsf{e[c]} \triangleq \wedge\ check.event.\mathsf{e} \\ \wedge\ check.condition.\mathsf{c}$$

$$check.event.\mathsf{[c]} \triangleq check.condition.\mathsf{c}$$

Validity of a set of event expressions is defined by

$$check.events : 2^{E_{expr}}$$
$$check.events(E_1) \triangleq \bigwedge_{\mathsf{e} \in E_{expr}} \mathsf{e} \in E_1 \Rightarrow check.event.\mathsf{e}$$

[13] This is the predicate *in*, (7)

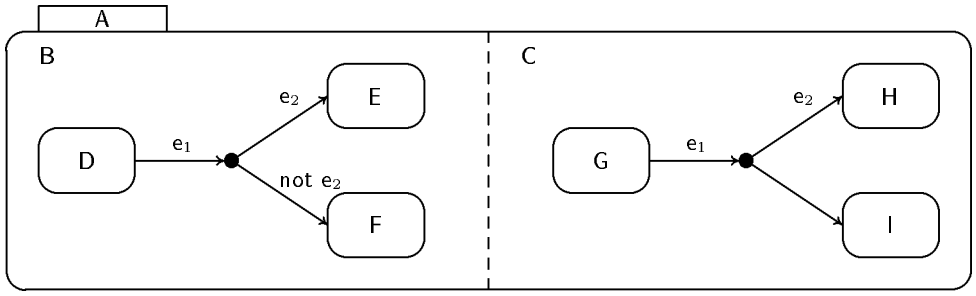


Figure 9: Effects of negated events

Whether a compound transition is enabled can be described by the following predicate, stating that it is actually a compound transition, all its source states are active, and all its event expressions are valid:

$$
\begin{array}{l}
compound.enabled : 2^T \\
compound.enabled(T_1) \triangleq \; \wedge\, one.compound(T_1) \\
\qquad\qquad \wedge \text{ All source states of the first part are active.} \\
\qquad\qquad\quad \bigvee_{T_2 \in 2^T} \; \wedge\, T_2 \subseteq T_1 \\
\qquad\qquad\qquad\qquad \wedge\, maximal.compound(T_2) \\
\qquad\qquad\qquad\qquad\quad \wedge \bigwedge_{t \in T_2} \bigwedge_{s \in S} s \in source(t) \\
\qquad\qquad\qquad\qquad\qquad\quad \Rightarrow in(s) \\
\qquad\qquad \wedge \bigwedge_{t \in T1} check.events(event(t))
\end{array}
$$

4.2.2 Composing a step

In one step, only enabled compound transitions can be taken whose scopes are not related hierarchically. An enabled compound transition is called *executable* if there is no other enabled compound transition with a strictly higher scope. Some of the nondeterminism between enabled transitions is resolved here with priority for transitions with a high scope. This scoping rule can be used too, to express priorities between transitions. And similar to negated events it destroys monotonicity: The existence of further events might enable a transition with a higher scope, as a consequence the one with the lower scope becomes unexecutable.

$$
\begin{array}{l}
compound.executable : 2^T \\
compound.executable(T_1) \triangleq \; \wedge\, compound.enabled(T_1) \\
\qquad\qquad\qquad\qquad \wedge \bigwedge_{T_2 \in 2^T} compound.enabled(T_2) \\
\qquad\qquad\qquad\qquad\quad \Rightarrow \neg substate^+(scope(T_2), scope(T_1))
\end{array}
\tag{8}
$$

A step of a statechart consists of a (non-empty) set of executable compound transitions with orthogonal scopes. Thus, each compound is independent of the other. These sets must be as large as possible with respect to subset inclusion. Thus, transitions in parallel components of a statechart are taken synchronously. When there are several maximal sets, which cannot be compared with respect to subset inclusion, one of them will be chosen nondeterministically.

$$
\begin{array}{l}
maximal.step : 2^{2^T} \\
maximal.step(\mathcal{T}_1) \triangleq \; \wedge\, \mathcal{T}_1 \neq \emptyset \\
\qquad\qquad\qquad\qquad \wedge \bigwedge_{T_1 \in \mathcal{T}_1} compound.executable(T_1) \\
\qquad\qquad\qquad\qquad \wedge \bigwedge_{T_1, T_2 \in \mathcal{T}_1} T_1 \neq T_2 \\
\qquad\qquad\qquad\qquad\quad \Rightarrow orthogonal(scope(T_1), scope(T_2)) \\
\qquad\qquad\qquad\qquad \wedge \bigwedge_{T_1 \in 2^T} T_1 \notin \mathcal{T}_1 \\
\qquad\qquad\qquad\qquad\quad \Rightarrow \neg maximal.step(\mathcal{T}_1 \cup \{T_1\})
\end{array}
\tag{9}
$$

In the statechart in figure 7, page 16, there are the compound transitions $\{t_1, t_2\}$ and $\{t_1, t_3\}$. When both of them are enabled then there is no further restriction which one to take in a

step, because they have the same scope. But it is not possible to take both of them. When regarding the statechart in figure 8 on page 16 it is assumed that all transitions are enabled. In the case, that control is in the states D and F, then the two compounds $\{t_1\}$ and $\{t_2\}$ can be taken simultaneously, because their scopes (the states B resp. C) are orthogonal. In the case {E,F}, the compound transition $\{t_3, d_2\}$ will be taken, but not the compound $\{t_2\}$. This is due to the fact, that the scope of the first one (Z) is higher than the scope of the latter one (C). In the case {E,G} only one of the compounds $\{t_3, d_2\}$ or $\{t_4, d_1\}$ can be taken in one step, because their scopes are not orthogonal.

When taking a step the set of active states, i.e. the configuration will change. All states strictly below the scope of one of the compound transitions are left and all states are entered which are strictly below the scope of one compound transition and above a target state (including them). As neither static reactions nor actions associated with exiting or entering states have been considered here, the set of actions to be performed is determined from the actions associated to the transitions. In this paper only the generation of new simple events is considered as a possible action. Hence it is sufficient to state that the set of events after taking a step is the union of the events generated by the actions of the transitions taken. Thereby all events existing prior to the execution of the step are discarded when taking this step.

$$action : T \rightarrow 2^E$$

$$take.step : 2^{2^T} \tag{10}$$

$$\begin{aligned} take.step^{14}(\mathcal{T}_1) \triangleq \; & \wedge \bigwedge_{s \in S} \; \wedge \; s \text{ is below the scope of a compound transition in } \mathcal{T}_1\text{:} \\ & \quad \bigvee_{T_1 \in \mathcal{T}_1} substate^+(scope(T_1), s) \\ & \quad \Rightarrow in'(s) \Leftrightarrow \bigvee_{T_1 \in \mathcal{T}_1} \bigvee_{s_1 \in S} \wedge \, s_1 \in targets(T_1) \\ & \qquad \wedge \, substate^*(s, s_1) \\ & \wedge \; s \text{ is not below the scope of a compound transition in } \mathcal{T}_1\text{;} \\ & \quad \neg \bigvee_{T_1 \in \mathcal{T}_1} substate^+(scope(T_1), s) \\ & \quad \Rightarrow in'(s) = in(s) \\ & \wedge \bigwedge_{e \in E} \; (\quad e \in E'_{exists} \Leftrightarrow \bigvee_{T_1 \in \mathcal{T}_1} \bigvee_{t \in T_1} e \in action(t) \quad) \end{aligned}$$

4.2.3 Execution of a statechart

Up to now only properties of a correct statechart and what it means to execute a step have been described. In the sequel the ongoing execution of a statechart is defined by a TLA-formula. The state variables S_{active} (5) and E_{exists} (6) have already been mentioned. Furthermore, a state variable *compound* will be used to record which maximal set of compound transitions has been taken in a step. This variable is also used to distinguish between TLA-steps generating new external events and those taking a step of the statecharts. This

[14] This is the first definition relating two different TLA-states, remember that primes are used to denote the variables in the new state.

will facilitate some of the proofs later on. All flexible variables are empty in the initial state. Furthermore it is required, that the statechart is a well formed one.

$$compound \subseteq 2^T \tag{11}$$

$$w \triangleq < S_{active}, E_{exists}, compound > \tag{12}$$

$$\begin{array}{l} initial.state : \\ initial.state \triangleq \begin{array}{l} \wedge\ well.formed \\ \wedge\ S_{active} = \emptyset \\ \wedge\ E_{exists} = \emptyset \\ \wedge\ compound = \emptyset \end{array} \end{array} \tag{13}$$

The sets of states, transitions, etc. are considered as rigid variables, hence it is not necessary to state that they do not change throughout a behavior.

Two TLA-actions are used to describe the next-state relation. One is corresponding to taking a step of the statechart. There will be no distinction between the first and subsequent steps executed. The initial condition implies that the first step will be a compound transition with an empty first part and a second part starting with the default transition of the root state. All subsequent steps will be 'normal' steps. If there are several possible steps in a statechart, then one of them will be chosen nondeterministically. The step chosen will be the new value of *compound*. This guarantees too, that not two maximal compound transitions can be taken in one step.

$$\begin{array}{l} step : \\ step \triangleq \bigvee_{\mathcal{T}_1 \in 2^{2^T}} \begin{array}{l} \wedge\ maximal.step(\mathcal{T}_1) \\ \wedge\ take.step(\mathcal{T}_1) \\ \wedge\ compound' = \mathcal{T}_1 \end{array} \end{array}$$

Note, that the change of the configuration is described by the predicate *take.step* (10). *maximal.step* (9) describes, which compound transitions are taken. By changing the definition *maximal.step* it is possible to relate other semantics of statecharts to the one defined here.

The second action is used to add further triggering events from outside of the statechart. This step sets *compound* to the empty set to make a clear distinction between this step and a *step*-step.

$$\begin{array}{l} new.event : \\ new.event \triangleq \begin{array}{l} \wedge\ E_{exists} \subsetneq E'_{exists} \\ \wedge\ S'_{active} = S_{active} \\ \wedge\ compound' = \emptyset \end{array} \end{array} \tag{14}$$

No fairness restriction is imposed on the action *new.event* since it may be infinitely often enabled without executing it. But weak fairness for the action *step* is required. This means, that if *step* is continuously enabled, then it will be taken eventually. This is a fairness

property which relates the TLA actions *step* and *new.event*, but it does not express any fairness property on possible steps of the statechart. The nondeterministic choice of enabled steps is not restricted and allows for neglecting one step as long as there is another enabled step. Also it is still possible to describe a statechart with a deadlock. In this case *step* cannot be enabled continuously and hence there must be no further progress.

Thus the possible behaviors of a statechart, given by the sets S, C, T, D, and E is described by the formula *statechart*.

$$\begin{array}{l} \mathit{statechart}: \\ \mathit{statechart} \triangleq \begin{array}{l} \wedge\ \mathit{initial.state} \\ \wedge\ \Box[\mathit{step} \vee \mathit{new.event}]_w \\ \wedge\ WF_w(\mathit{step}) \end{array} \end{array} \tag{15}$$

In this formula all variables are visible. All, except the variables defining the behavior (S_{active} and E_{exists}) and the sets necessary to reason about them (E and S), should be hidden. Following [Manna and Pnueli, 1992] this will be called the *temporal semantics* of a statechart[15].

$$\begin{array}{l} \mathit{stc}: \\ \mathit{stc} \triangleq \exists\, C, T, D : \exists\, \mathit{compound} : \mathit{statechart} \end{array} \tag{16}$$

This temporal semantics is simple, but it is appropriate for the subset of statecharts used here. Before examining some variations of the semantics in the next section, it will be clarified, that the temporal semantics actually captures the behaviors generated by the statechart. At first it will be shown, that in each state of a behavior satisfying *stc* (16), the configuration of states is a correct one. At second, some implications of the fact, that different steps having the same effects cannot be distinguished and the relation of stuttering steps of behaviors satisfying *stc* with the corresponding statechart are examined.

It is clear, that in each state of a behavior satisfying *stc*, the sets of states, transitions, etc. form a well formed statechart. This is required in the initial state – by the predicates *initial.state* (13) and *well.formed* (1) – and these sets are not changed throughout a behavior. It is easy to see, that in the initial state of a behavior satisfying *stc* the set S_{active} is a correct configuration, i.e. all direct descendants of an active *and*-state and exactly one direct descendant of an active *or*-state are active too. This is vacuously valid, because there is no active state in the initial state of such a behavior, see the definition of *initial.state*. The initial configuration of the statechart is entered explicitly by a compound transition. This is a compound transition with an empty first part, extending the root state towards the *basic*-states. As a part of the predicate characterizing compound transitions – *one.compound* (4) – it is required that the *basic* target states of a compound transition are a correct configuration, see the definition of *correct.target* (3). By taking a compound transition as part of a step a state below (and including) the scope of the transition becomes active if and only if it is above a target state of the transitions, again including them. Because the scope of this first transition is the *root*-state itself, a correct configuration of the complete statechart is entered. Note, that if no such transition exists, because e.g. either it is not possible to construct a correct compound transitions or because a default transition to be used has a triggering event

[15] In the following the existential quantification of C, T and D will be omitted to simplify the formulae.

or condition which is not valid, then no initial configuration is reached and the *root*-state is not entered in a behavior satisfying *stc*. But in the statechart it will be not possible also to determine its initial configuration in such a case.

Similarly it can be argued, that this property is preserved by taking an ordinary *step*-step. Below the scope of each compound transition taken in this step, a correct configuration is achieved. States above or besides the scopes of the compound transitions remain (in-) active, hence the new set of active states is again a correct configuration of the complete statechart.

Steps of the statechart have been described by one TLA-action, hence no intermediate states, which do not correspond to configurations of the statechart are visible. To achieve such a description has not been difficult here, because the effects of taking a step are simple. But when considering more general actions, this description of a step will become more complicated.

It has been argued above, that steps of the statechart are modeled correctly by the temporal semantics and hence only correct configurations can be reached. On the other hand, it has to be shown, that a behavior satisfying *stc* corresponds to a behavior of the statechart.

Remember that in a behavior of a statechart, as well as in a behavior satisfying *stc*, in each state the set of active states and the actual events are visible. It has to be shown only, that given a behavior satisfying *stc*, there is a corresponding sequence of states of the statechart, but it is not required, that the same sequence of steps is taken in the statechart. A first difference between the two kinds of behaviors is, that the one satisfying *stc* may contain stuttering steps and is always an infinite behavior, whereas the relation between subsequent states of a behavior of a statechart is expressed by an explicit step of the statechart or by adding further events and a behavior can have finite length. But because no notion of time has been considered, neither in the statechart nor in the temporal semantics, stuttering steps can be considered simply as looking repeatedly at the same state of a statechart. Infinite sequences of stuttering steps delaying real *step*-steps have been excluded by the fairness property on the TLA-action *step*, hence a behavior satisfying the temporal semantics and which is stuttering continuously from a certain state is corresponding to a behavior of statechart reaching a state which will never be left. Note, that there are possible steps of the statechart which do not change the configuration and which reproduce the set of events. This steps cannot be distinguished from stuttering steps in a behavior satisfying the temporal semantics. In a behavior of the statechart this steps appear as repeated states. Because enabledness of transitions is not changed by such steps, it does not cause any problems that this distinction cannot be made. It is also the case, that steps extending the set of events without changing the configuration cannot be distinguished from externally adding further events, but again, this situation arises similar in the temporal semantics as well as in a behavior of a statechart.

In the semantics presented here, only one statechart has been considered. It is clear, that if several statecharts are considered and their steps have to be synchronized, then it has to be made visible, that a *step*-step is taken instead of a stuttering step.

4.3 Comparison with other statechart variants[16]

The temporal semantics defined here is similar to that proposed by Day et.al. [Day, 1993, Day and Joyce, 1993]. But as this one, it does not support the synchrony hypothesis, see

[16] An overview on existing variants of statecharts and their semantics is given in [van der Beeck, 1994].

e.g. [Berry and Gonthier, 1992], which states, that reaction upon external stimuli is instantaneous. This means especially, that the reaction upon one stimulus has taken place before the next stimulus arrives at the system. When several steps of a statechart are needed to determine the reaction upon a stimulus, then in the semantics used here this might interfere with new stimuli arriving. Already in the small example presented later on this will lead to a smaller problem when considering a refinement relation.

As an alternative the original semantics of statecharts [Harel *et al.*, 1987], which has been improved in [Pnueli and Shalev, 1991], can be considered. There, a step of a statechart consists of several so called *microsteps*. Each microstep corresponds to taking one transition. Starting from one executable compound transition, a maximal set of compound transitions is constructed successively. Throughout this construction process, enabledness of transitions is evaluated with respect to the events existing at the beginning of the step and the events generated by the transitions which have been considered already. Because all compound transitions considered in one step have to be non-conflicting as described earlier, the construction process is terminating. In the improved version of the semantics [Pnueli and Shalev, 1991], the sets of compound transitions which can be constructed in this way can be characterized by a consistency condition on compound transitions. Using this condition in the definition of *maximal.step* (9), this semantics can be embedded seamlessly in the temporal semantics presented above. Because it is possible to take several causally related compound transitions in one step, which can also be considered as a chain of reactions, the synchrony hypothesis is supported in this semantics to a considerable larger extend than in the semantics used here.

As pointed out in [Leveson *et al.*, 1994] this construction may lead to steps, which has been classified there as counterintuitive: It is possible, that compound transitions, which are executable at the beginning of a step are not executed in it, even if there are no other ones with the same scope initially. Hence, in the statechart variant RSML, repeatedly a maximal set of compound transitions is taken as a response to an external stimulus. This is repeated, until there are no more executable transitions. Enabledness of the intermediate transitions is based only on the events generated by the transitions just taken. New external events are recognized only after finishing such a step, which will be called a *superstep* according to [stm91, 1991]. This semantics is more intuitive and it still maintains the synchrony hypothesis, but it is very easy to construct statecharts with nonterminating steps. This semantics can also be embedded in the temporal semantics presented above, by restricting the TLA-action *new.event*: This action should be enabled only when no compound transition is enabled.

5 An example: A lift controller

In this section a small, but nontrivial, example is presented: a lift shall be controlled. Thereby it is intended to show, how the proposed combination of formalisms could look like and how it could be used.

At first, the plant is described from the view of a potential user. In this example this can be done conveniently by a statechart without using temporal logic[17]. The set of possible behaviors of the plant is restricted then by several requirements. These are stated in natural language and will be formalized by changes to the statechart describing the plant and by imposing further temporal properties on the behaviors of the plant. This section finishes

[17] At the end of this section a variant of the plant is described using an explicit liveness property.

with some remarks on the formalizations given in this section.

5.1 The plant

It is possible to view the plant from several different viewpoints. In this example, the viewpoint of an external or potential user of the lift and the viewpoint of a controller can be distinguished. The main difference is, that from the viewpoint of the external observer a part of the behavior of the lift seems to be nondeterministically, e.g. the lift cabin can stay on a floor or it can move to another one. From the viewpoint of the controller this behavior is deterministic, because the controller actually controls this movement. In the following, the plant is described from the viewpoint of the external observer, the viewpoint of the controller can be obtained by additional events and conditions on some of the transitions. These additional events are shown in the statechart in figure 17, page 45.

It is assumed to have a building with n floors and one lift, see the statechart in figure 10. The lift itself is on exactly one of the floors of the building in each state. In each step the lift could move to the next floor in either direction or it could stay on the actual one. The amount of time necessary to move the lift physically is abstracted away. An user could request the lift by pressing a push-button on a floor or pressing one inside the lift cabin. These cases are not distinguished in this model of the plant, the lift is requested to move to the floor i simply by generating the event press_i. A request is considered served, when the door on the corresponding floor opens. Initially, the lift is on the first floor, all doors are closed and there is no request. The semantics of this statechart can be described in the same way as the

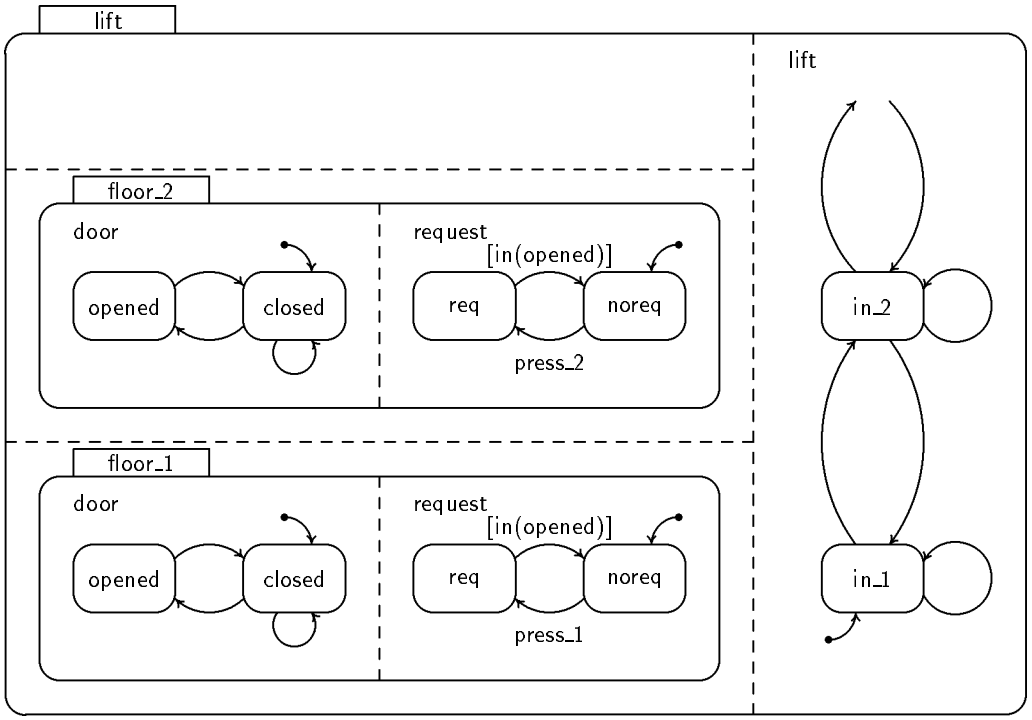


Figure 10: Lift example, the model of the plant

predicate *statechart* (resp. *stc*) (15) on page 28:

$$
\begin{array}{l}
plant: \\
plant \triangleq \ \wedge\ initial.state \\
\qquad\quad \wedge\ \Box[step \vee new.event]_w \\
\qquad\quad \wedge\ WF_w(step)
\end{array}
$$

Note, that the nondeterministic behavior is modeled explicitly by several transitions leaving the same states which can be taken in the same configuration.

5.2 The specification

Some reasonable requirements on this lift, stated in natural language, are:

1. The door on a floor is opened only if the lift is on the same floor.
2. The lift may leave a floor only if there is no request for this floor.
3. All requests are served.
4. The door is opened on a floor only if there is a request.
5. The lift changes its direction of moving only if there is no further request in the same direction.
6. The lift makes no unnecessary moves.
7. The lift does not remain on a floor, while there are other requests.

In the following these requirements will be captured formally. They will be captured one after the other and combined at the end of this section. The main purposes of this formalization are to capture the requirements more precisely than is possible by the use of natural language and to provide a basis for formal proofs of the correctness of designed controllers.

In this paper, each of the requirements will be captured by changing the description of the plant, either by changing it explicitly, by conjoining TLA-formulae to it, or implicitly, by changing the statechart. In both cases it is checked, whether at least each individual requirement is satisfiable. But it is clear, that this does not imply satisfiability of all requirements together. Furthermore, especially when changing the plant implicitly, the relationship between the description of the plant and the requirement has to be made clear. It has to be guaranteed, that each behavior satisfying the requirement is a possible behavior of the plant. This means formally: the description of a requirement is a refinement of the description of the plant. Let *requirement* denote the TLA-formula describing a requirement, then this is expressed in TLA by $requirement \Rightarrow plant$, whereby $\Rightarrow$ is logical implication.

5.2.1 Adding a safety property

The first of the requirements is a typical safety property, both in an informal as well as in a formal sense: If it is not fulfilled, the lift may be in an unsafe state for its users. Formally, it is a safety property according to the definition of [Manna and Pnueli, 1992] as a (past)

formula to be valid in all states. More precisely, it is an invariance property because it refers to the current state only. This requirement can be expressed by the predicate *safe.door* and by requiring that *safe.door* is valid in all states.

$$
\begin{array}{l}
safe.door: \\
safe.door \triangleq \bigwedge_{1 \le i \le n} in(\mathsf{floor\_}i\mathsf{.door.opened}) \Rightarrow in(\mathsf{in\_}i)
\end{array}
$$

$$
\begin{array}{l}
requirement.1: \\
requirement.1 \triangleq \; \wedge\, plant \\
\qquad\qquad\quad \wedge\, \Box safe.door
\end{array}
$$

[18]

It is clear, that conjoining a temporal formula, which refers only to the flexible variables S_{active}, E_{exists}, and *compound*, to *plant* results in a refinement of *plant*. It is also easy to see, that there is at least one behavior satisfying this safety requirement: In this behavior all doors are kept closed[19]. But there are also more useful behaviors satisfying this requirement.

requirement.1 is not a TLA-formula in canonical form, but it can be shown, that it is equivalent to one. In the following, *requirement*.1 is reformulated such that *safe.door* is implied by the initial condition and that taking a step preserves this property[20].

The definition of *safe.door* is based on the variable S_{active} (via the predicate *in*). Because S_{active} is part of the state function w, it is easy to see that stuttering steps preserve the property *safe.door*:

$$
\begin{array}{l}
\wedge\, safe.door \\
\wedge\, w = w' \\
\Rightarrow\, safe.door'
\end{array}
$$

Hence the inference rule TLA.1 can be applied to get

$$
\begin{array}{l}
\Box safe.door \equiv \wedge\, safe.door \\
\qquad\qquad\quad \wedge\, \Box[safe.door \;\Rightarrow\; safe.door']_w
\end{array}
$$

and

$$
\begin{array}{l}
requirement.1 \equiv \wedge\, \wedge\, initial.state \\
\qquad\qquad\qquad\quad\; \wedge\, safe.door \\
\qquad\qquad\qquad \wedge\, \wedge\, \Box[step \vee new.event]_w \\
\qquad\qquad\qquad\quad\; \wedge\, \Box[safe.door \;\Rightarrow\; safe.door']_w \\
\qquad\qquad\qquad \wedge\, WF_w(step)
\end{array}
$$

[18] The parts of the requirements which are changed with respect to the description of the plant are highlighted.

[19] It is clear, that this behavior does not satisfy requirement 3.

[20] The rest of this paragraph can be skipped on first reading.

which is equivalent to

$$
\begin{aligned}
requirement.1 \equiv \; & \wedge \; initial.state \wedge safe.door \\
& \wedge \; \Box \left[\begin{array}{l} \vee \wedge \; step \\ \quad \wedge \; safe.door \; \Rightarrow \; safe.door' \\ \vee \wedge \; new.event \\ \quad \wedge \; safe.door \; \Rightarrow \; safe.door' \end{array} \right]_w \\
& \wedge \; WF_w(step)
\end{aligned}
\tag{17}
$$

This is already a TLA-formula in canonical form, but in this case it can be simplified further. The action *new.event* does not change the flexible variable S_{active} and thereby validity of *safe.door*, hence it is only necessary to restrict the action *step*.

$$new.event \; \Rightarrow \; (safe.door \; \Rightarrow \; safe.door')$$

implies

$$
\begin{aligned}
new.event \equiv \; & \wedge \; new.event \\
& \wedge \; safe.door \; \Rightarrow \; safe.door'
\end{aligned}
$$

Similarly $initial.state \; \Rightarrow \; safe.door$ and hence *requirement*.1 is equivalent to the following formula:

$$
\begin{aligned}
& \wedge \; initial.state \\
& \wedge \; \Box \left[\begin{array}{l} \vee \wedge \; step \\ \quad \wedge \; safe.door \; \Rightarrow \; safe.door' \\ \vee \; new.event \end{array} \right]_w \\
& \wedge \; WF_w(step)
\end{aligned}
\tag{18}
$$

5.2.2 Constraining transitions

The second requirement states, that the lift should not leave a floor, as long as there is a request for this floor. Therefore, the nondeterminism between the transitions leaving one of the states in_*i* and those reentering them has to be resolved. In resolving this nondeterminism, again the view of an external observer is taken. It is clear, that implicitly the occurrence of the events used to control the movement of the lift, see figure 17, will be restricted, too.

To express the requirement it is not necessary to resolve the nondeterminism completely, only the undesired behaviors have to be excluded. It can be expressed both by adding conditions to appropriate transitions of the statechart as well as by restricting the TLA-action *step*.

Constraining transitions of the statechart: The requirement can be expressed by adding conditions to the transitions from a state in_*i* to a state in_*j* with $i \neq j$. These conditions express that there must not be a request for floor *i* in the current configuration of the statechart, see figure 11. Note, that it is allowed that a state in_*i* is left while the event press_*i* is currently existing. Hence it is not possible to hold fast the lift on a floor by requesting the lift without interrupt.

By adding these conditions to the statechart, the formula *plant* denoting the temporal semantics of the plant has been changed implicitly. Let *requirement*.2 denote the formula denoting the temporal semantics of the statechart given in figure 11. It has to be shown that *requirement*.2 is a refinement of *plant*. Because the visible sets of active states and existing events of *requirement*.2 are the same as those of *plant* it is possible to relate their behaviors. It is also clear, that a *step*-step changing the position of the lift – $in(\mathsf{in\_}i) \wedge \neg in'(\mathsf{in\_}i)$ – of *requirement*.2 is also a possible step of *plant*. Let us assume now, that the lift is on floor i and that the condition reqi is valid. Hence, in a step of a behavior satisfying *requirement*.2 it is not possible that the lift leaves this floor. But because the self loop from the state in_i to itself can be taken unconditionally, this will – and has to – be taken in this step. Other compound transitions taken in this step may influence only the states floor_*i*. It is also possible to take a *step*-step consisting of these transitions and the self loop in a behavior satisfying *plant*, hence *requirement*.2 $\Rightarrow$ *plant*.

Constraining the action *step*: The requirement can also be expressed by changing the formula *plant*, describing the temporal semantics of the statechart, explicitly. The following TLA-action *no.move.i* expresses, that when the states in_*i* and floor_*i*.request.req are active, then the state in_*i* will be still active after taking one step.

$$
\begin{array}{l}
no.move.i: \\
no.move.i \;\triangleq\; \wedge\; in(\mathsf{in\_}i) \\
\qquad\qquad\;\; \wedge\; in(\mathsf{floor\_}i.\mathsf{request.req}) \\
\qquad\qquad\;\; \Rightarrow\; in'(\mathsf{in\_}i)
\end{array}
$$

For each of the floors, this predicate is conjoined to the action *step*. It is not necessary to constrain the action *new.event*, because this does not influence the activity of states.

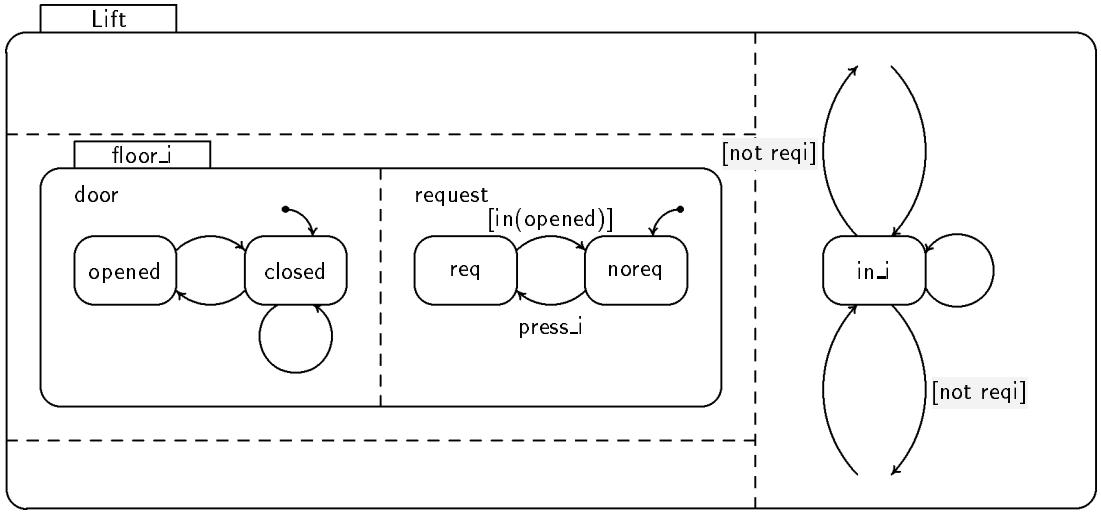


Figure 11: Lift example, constraining transitions

$$
\begin{array}{l}
requirement.2a: \\
requirement.2a \triangleq \ \wedge\ initial.state \\
\qquad\qquad\qquad\qquad \wedge\ \Box \left[\begin{array}{l} \vee \wedge\ step \\ \quad \wedge \bigwedge\limits_{1 \le i \le n} no.move.i \\ \vee\ new.event \end{array} \right]_w \\
\qquad\qquad\qquad\qquad \wedge\ WF_w(step)
\end{array}
$$

Again it is easy to see, that there is at least one (useful) behavior. Also, it is not necessary to change the fairness property of *step*: Whenever a *step*-step is possible, then one obeying the restriction can be taken too: In each step one of the transitions from a state in_*i* to itself can be taken. This implies, that the specification remains machine closed[21] and hence no new safety property is introduced. Furthermore, trivially each restricted *step*-step is also an unrestricted one and hence *requirement*.2*a* is a refinement of *plant*.

5.2.3 Adding a liveness property

Requirement 3 is a liveness property, stating that something good will happen eventually, viz each request will be served. But it is not required here, that the request is served as soon as possible. The requirement can be expressed in (at least) three different ways: as a fairness property, by a leadsto formula, and by helpful transitions.

Adding a fairness property: Serving a request has been defined as opening the door on the corresponding floor. This can be expressed by a restricted version of the TLA-action *step* corresponding to the transition between the states floor_*i*.door.closed and floor_*i*.door.opened:

$$
\begin{array}{l}
grant.i: \\
grant.i \triangleq \ \wedge\ step \\
\qquad\qquad \wedge\ in(\mathsf{floor\_}i\mathsf{.request.req}) \\
\qquad\qquad \wedge\ in(\mathsf{floor\_}i\mathsf{.door.closed}) \\
\qquad\qquad \wedge\ in'(\mathsf{floor\_}i\mathsf{.door.opened})
\end{array}
$$

These TLA-actions are enabled, when *step* is enabled and there is an unserved request. The requirement itself is captured by requiring weak fairness of these actions: If one of these actions is enabled continuously, it has to be taken eventually, thereby serving the request. Because these actions remain enabled until they are taken, it is sufficient to require weak fairness instead of strong fairness. This fairness property implies, that serving a request can be delayed for only a finite number of *step*-steps.

As before, the formulae *grant*.*i* can be conjoined to the formula *plant*:

$$
\begin{array}{l}
requirement.3: \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (19) \\
requirement.3 \triangleq \ \wedge\ initial.state \\
\qquad\qquad\qquad\quad \wedge\ \Box[step \vee new.event]_w \\
\qquad\qquad\qquad\quad \wedge\ WF_w(step) \\
\qquad\qquad\qquad\quad \wedge \bigwedge\limits_{1 \le i \le n} WF_w(grant.i)
\end{array}
$$

[21] A specification is called *machine closed*, if its liveness properties do not imply further safety properties, see [Abadi and Lamport, 1991] and [Abadi and Lamport, 1994].

Because each *grant.i*-step is also a *step*-step no new safety properties are introduced, and one has $requirement.3 \Rightarrow plant$.

Adding a leadsto formula: The formulation of this requirement in natural language can be formalized directly as a leadsto-formula.

$$\begin{array}{l} serve.i: \\ serve.i \triangleq in(\mathsf{floor\_i.request.req}) \leadsto in(\mathsf{floor\_i.door.opened}) \end{array} \tag{20}$$

The formula $in(\mathsf{floor\_i.request.req})$ remains valid until the request is served. This implies, that when the TLA-action *step* is enabled, and the request is not served, then the TLA-action *grant.i* is enabled too. By $WF_w(grant.i)$ it can be concluded that *grant.i* is taken eventually, which implies $in(\mathsf{floor\_i.door.opened})$. Hence the leadsto-formula (20) is implied by the fairness formulae (19). As (19) is machine closed, this is also true for $plant \wedge serve.i$, see proposition 3 in [Abadi and Lamport, 1994, page 1551]. Similarly it can be concluded, that this is a refinement of *plant* and that the set of behaviors satisfying this requirement is not empty.

Using a helpful transition: Alternatively, requirement 3 can be expressed as the conjunction of two other requirements: The lift eventually reaches each floor, for which there is a request. And, if the lift is on a requested floor, this request will be served eventually. The first one is requirement 7, the second one will be expressed here using *helpful transitions*[22]. These are defined as transitions, which bring a behavior nearer to a specific goal. This notion is used here to resolve nondeterminism between transitions of the statechart. This is resolved such that it is not possible to keep the door closed on a floor, when there is a request for this floor and the lift is on this floor.

The states closed should not remain active if it is possible to serve a request. Hence, the transition from closed to closed should be taken only if it is not possible to serve a request. The appropriate condition is not (in(req) and in(in_i)) for each floor. Now the transition from the state closed to the state opened must be taken, see figure 12. In this statechart, the transition has been restricted in addition with the condition in(req), which expresses requirement 4. Arguing as above it can be concluded, that expressing the requirement in this way gives a refinement of the description of the plant.

Using this kind of formalization it can be expressed, that a request should be served as soon as possible, whereas the leadsto- and the fairness-formulae are only able to express that the request should be served eventually. But one should be aware that this is a very stringent form of bounded response, which could be in conflict with other requirements. Note, that a bounded response property is a safety property. Henceforth it is also possible here to express these helpful transitions by restricting the TLA-action *step* directly.

The nondeterminism between the transitions leaving the states closed has not been resolved completely. If the lift is not on a requested floor it is both possible to open the door or to keep it closed. But the first behavior is excluded by property 1, therefore it is not necessary to take it into account here.

[22] See e.g. [Manna and Pnueli, 1993].

5.2.4 Extending the model of the plant

Requirement 5 refers to the direction of the movement of the lift. When specifying this requirement, the direction can be taken into account as a past formula in the sense of [Manna and Pnueli, 1992] relating the actual floor the lift is in and the most recent different one. Another possibility would be to define a history dependent variable as proposed in [Abadi and Lamport, 1994] to remember the trace of movements or just the direction of the last move. Here another approach will be followed, because the direction of the movements of the lift can be seen as a state of the lift. Therefore a further *and*-state direction is added to the statechart describing the plant to store the direction of movement. This state stores not only the direction of the last movement, it also indicates the direction of the next one according to the requirements.

The state direction has two descendants: one intended for a stationary lift when there are no requests and one intended for a moving or requested lift. This last one has two substates indicating the direction of movement. Initially the lift is considered stationary. See figure 13 for a statechart describing this state including the transitions. The state move is left if there are no requests. The appropriate descendant of it is entered if a new request is entered after some period in which the lift has been stationary. Changes between the states up and down are made according to the requirements. Note, that in the formulation of requirement 5 in natural language it does not become clear into which direction the lift should move after serving all requests if there are simultaneously new ones in both directions. The lift could move in the direction of the last movement or it can move in either of these directions because the movement has been interrupted. In the statechart in figure 13 the second choice has been modeled, because there is no reason why the lift should keep its direction of traveling after all pending requests have been served.

Up to now this new state captures only the notion of direction and when this is allowed to change. The actual movement of the lift has to correspond to the substates of the state move.

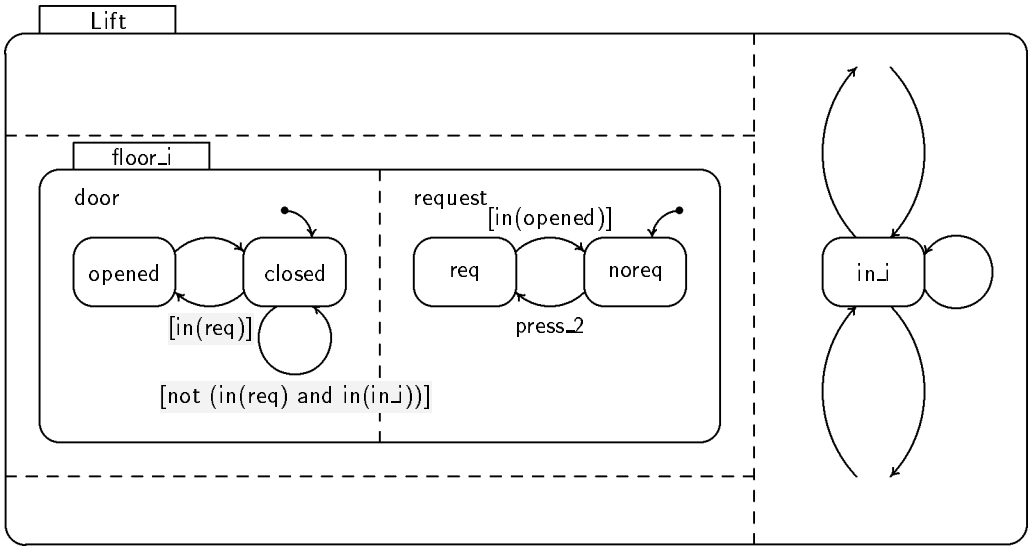
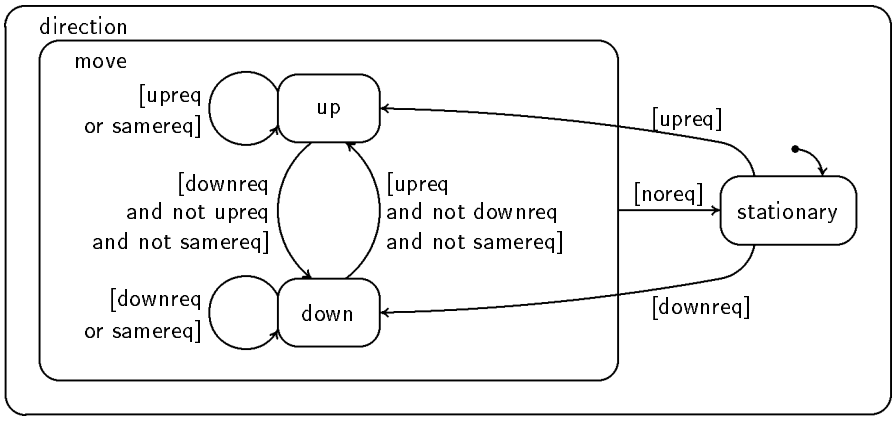


Figure 12: Lift example, using helpful transitions

This can be done easily by constraining the transitions between the states in_*i*, see figure 14. The behaviors of the plant and of this extension cannot be compared directly. The behaviors of the extension have to be restricted to the set of states of the plant. By simply adding this extension, no new behavior is introduced, because steps of the plant are independent of those of the extension. Furthermore, the extension cannot make a step without the plant making a step, too. Hence it can be shown, that the extended plant, restricted to the appropriate sets of observable behaviors is a refinement of the plant.

5.2.5 Restricting the actions

By the extension of the statechart shown above, behaviors which are forbidden by requirement 6 have already been excluded implicitly: If there is no request, then the state stationary becomes active and the lift cannot move. But this is valid only in combination with requirement 1. The lift is ready to move only when the door has been closed after serving a request.



The condition upreq is true if there is a request strictly above the actual position of the lift, downreq is defined similarly. samereq is true if there is a request on the floor the lift is actual on. And noreq is defined as not (samereq or upreq or downreq).

Figure 13: Lift example, extending the plant

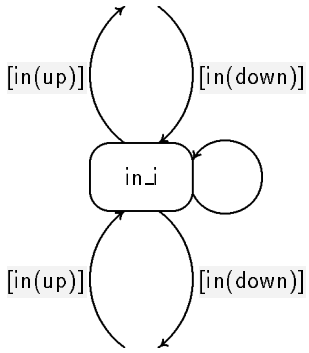


Figure 14: Lift example, constraining the movement of the lift

To do this, more steps have to be taken in the model used here than are necessary to recognize that there is no request and to enter the state stationary. But whenever changing the formulation of requirement 5 it is necessary to check whether requirement 6 is still satisfied.

This requirement can be expressed more explicitly by restricting the TLA-action *step*. To do this the action *staying* will be conjoined to it.

staying :

$$staying \triangleq check.condition.\mathsf{noreq} \Rightarrow \bigwedge_{1 \le i \le n} in(\mathsf{in\_}i) = in'(\mathsf{in\_}i)$$

It is not necessary to conjoin *staying* to the TLA-action *new.event* because this does not change the set of active states. It is also possible to express this requirement by restricting the transitions between the states in_*i* with the condition not noreq.

The last requirement (7) is essentially a liveness condition, or taken with a stronger meaning it is a bounded response property. Taken as a liveness condition, it is subsumed by requirement 3, taken as a bounded response property it can be treated in the same way as this requirement.

5.3 Combining the properties

When combining the requirements, the formulations using fairness of the requirements 3 and 7 have been used. The overall specification consists of the original statechart and its extensions. Several restrictions on the same transition are combined by conjunction, see figure 15. The temporal semantics of this statechart together with the additional properties is defined by the formula *lift*:

lift :

$$\begin{aligned} lift \triangleq\ & \wedge\ initial.state \\ & \wedge\ \Box \left[\begin{array}{l} \vee\ step \wedge staying \\ \vee\ new.event \end{array} \right]_w \\ & \wedge\ \Box safe.door \\ & \wedge\ WF_w(step) \wedge \bigwedge_{1 \le i \le n} WF_w(grant.i) \end{aligned}$$

It has been shown, that each single requirement is a refinement of the description of the plant. But it is not as easy to show, that there is at least one behavior satisfying all requirements. E.g. the requirements 2 and 3 are almost contradicting: Suppose, the lift is requested continuously for one floor and the lift cabin is on this floor, then there is exactly one possible step[23] after serving a request to move the cabin to another floor.

To cope with the problem, that neither the additional states nor the internal events used to control the plant should be visible in the specification, two flexible variables holding the sets of visible states and events will be introduced, whereas the variables S_{active} and E_{exists} will become hidden variables.

[23] This tight coupling will give further problems later on when showing correctness of a controller with respect to the specification.

Let S_{plant} be the set of states defined in the statechart describing the plant, as shown in figure 10. The flexible variable S_{obs} is defined to contain the set of observable states. It is related to S_{active} by the formula $S_{obs} = S_{active} \cap S_{plant}$, which must be valid in all states. Correspondingly, let *external.events* be the set $\{\mathsf{press\_}i \mid 1 \leq i \leq n\}$. The set of observable events E_{obs} is defined as $E_{obs} = E_{exists} \cap external.events$. Thus the final specification[24] is expressed by

[24]The 'internal' variables (*compound*, C, T, D) are not hidden explicitly to simplify the formula.

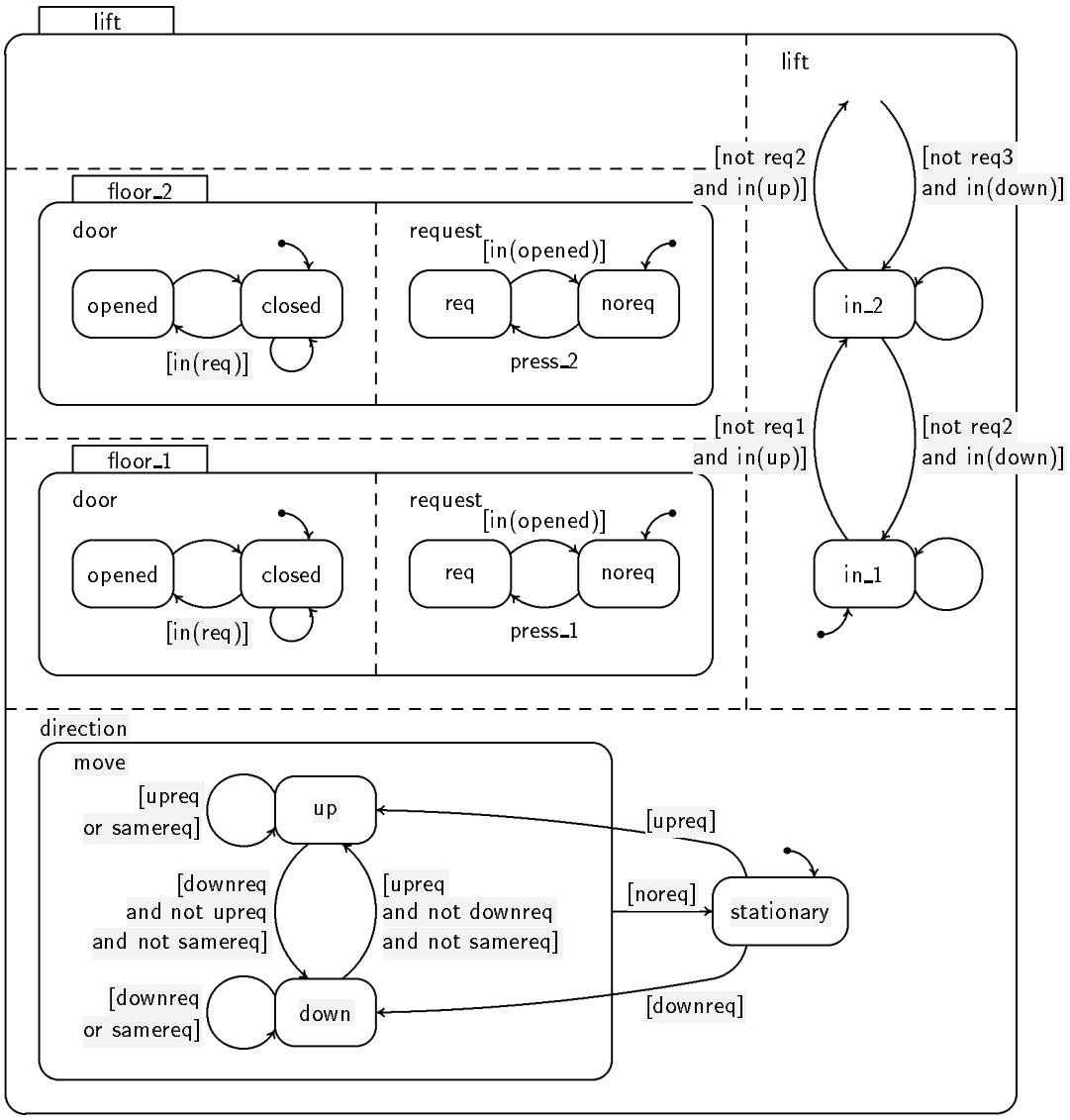


Figure 15: Lift example, the statechart of the specification

$$
\begin{array}{l}
lift.final: \\
lift.final \triangleq \exists\, S_{active}, E_{exists} : \wedge\ lift \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \Box S_{obs} = S_{active} \cap S_{plant} \\
\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \Box E_{obs} = E_{exists} \cap external.events
\end{array}
\tag{21}
$$

It has to be checked that *lift.final* is a refinement of *plant*. If two requirements are described by restricting *plant* explicitly, then it follows immediately that the combination of them, which is simply their conjunction, is again a refinement of *plant*. When the requirements are expressed implicitly, then combining them means to define a new statechart. But it is not necessarily the case, that the formula describing its temporal semantics is a refinement of *plant*. It has to be investigated further, under which conditions on the implicit formulations of requirements it is guaranteed, that their combination is a refinement of the plant. Also it is clear, that the combination of consistent specifications does not need to be consistent.

5.4 Some conclusions

5.4.1 When should one use which language?

It has been possible to express the same requirement in different ways and using different languages. Naturally, the question arises, which requirement should be expressed in which language?

It has been convenient here to express the safety requirement and the liveness requirements using temporal logic. The formulation of these requirements is understandable and precise. Adding the temporal formulae to the description of the plant can be done without problems.

On the other hand, it has been convenient to abstract the history of the system into additional states. This has been done when describing the control strategy of the movement of the lift, which is part of the specification. The pure extension did not influence the description of the plant, therefore the set of behaviors (restricted to the states of the plant) was the same as the set of behaviors of the plant. The extension can be connected to the plant by the same methods used to describe other properties. Using past formulae or history dependent variables would give reformulations of the state-based formulation of this property in the corresponding temporal logic. Because the direction of the lift can be expressed in such a clear way by states, the graphical formulation is considered more convenient than one by a formalism based on logic. But it should be clear, that this is a rather subjective evaluation.

When restricting transitions it is less clear, which language should be used. In the example TLA-formulae as well as additional conditions for the transitions of the statechart have been used. Up to now we cannot give hints which kind of formulation is preferable.

In this example there are two patterns to restrict transitions. These patterns appear already in the formulation of the requirements in natural language. Let *pre* and *post* be conditions refering to the source and target states, resp., of one transition and let *cond* be a condition on the state of the statechart, then the patterns are: If the system is in a *pre*-state and *cond* is valid, then the next state shall be a *post* state. And, a *post*-state may be reached only from a *pre*-state, if *cond* is valid. Expressed in logic, these are $pre \wedge cond \Rightarrow post'$ and $pre \wedge post' \Rightarrow cond$. Although, the two patterns are related trivially in logic, $(pre \wedge cond \Rightarrow post') \Leftrightarrow (pre \wedge \neg post' \Rightarrow \neg cond)$, they are used in natural language with a certain intention. The first one is used to express progress in a specific direction, whereas the second one excludes

transitions to undesired states.

In statecharts it is possible only to constrain transitions by conditions, it does not make sense to enable them explicitly, because a transition is already enabled if the associated condition is valid. Therefore to express the first pattern, one has to use the logically equivalent form and restrict all transitions from *pre*- to non-*post*-states with the negated condition.

Also, there is no general rule, whether it is better to conjoin a temporal formulae to the TLA-action *step* or whether it is better to extend the enabling conditions of transitions. Adding conditions uses the advantages of the visual notation which can make it easier to understand implications of the requirement on the plant. But the formulation of the property is more implicit and more dependent on the step semantics of the statechart which makes it more intricate to show that the description of a requirement is a refinement of the plant. On the other hand, by restricting the action *step*, a property can be expressed more explicitly.

Furthermore, when restricting steps of the statechart, either implicitly or explicitly, it should be ensured, that *step* remains a subaction of the restricted steps. This means, that it should always be possible to take a transition in each parallel component, which are in the example used here the direct descendant of the *and*-states. Otherwise further safety properties might be expressed unintentionally.

Therefore it is proposed to formulate safety and liveness (as fairness) properties explicitly by TLA-formulae. If bounded liveness has to be described, this can be done using helpful transitions. Adding constraints to transitions should be done reluctantly and very carefully to avoid that the plant shows new behaviors. Extensions of the statecharts can be used to abstract the history of the system into states and to describe control strategies which are part of the requirements.

5.4.2 On the formulations of plant and requirements

When looking more closely at the description of the plant, one can see, that some of the techniques used in the formulation of the requirements can be used here, too.

In the description of the plant, see figure 10, the states req and noreq of each floor have been used to indicate that the lift is requested. These states may have physical counterparts in the real world, such as lights which are on if the lift is requested. But these states can also be seen as an extension of the plant which helps to describe properties. In our case it is easier to refer to these states, than to the events press_*i* to express, that the lift had been requested, but had not yet served this request. These additional states can be seen as an abstraction of the history of the system, similar to the modes in the work of Parnas et.al. [Courtois and Parnas, 1993, van Schouwen *et al.*, 1993].

Up to now, temporal logic has not been used in the description of the plant, except when defining the temporal semantics of the statechart describing it. Therefore, it can be asked whether this combined language is useful for the description of the plant or whether it just adds additional complexity which should be avoided in practice. Using a slight variation of the description of the plant, it can be seen, that temporal logic can be used conveniently already in the description of the plant. In the statechart in figure 10 it is described, that an open door closes in the next step of the statechart. The model of the lift might be over-simplistic here, it could be more realistic to state, that the door may be open for some time, but eventually it will close. To express this property, the state door has to be changed for

each floor as can be seen in figure 16: The door can be closed or it may remain open in the next step. Closing the door on floor i corresponds to the TLA-action $close.i$. By requiring weak fairness for these actions, the property can be expressed, yielding a new description of the plant:

$$\begin{aligned} close.i \;\triangleq\; & \wedge\ step \\ & \wedge\ in(floor\_i.door.opened) \\ & \wedge\ in'(floor\_i.door.closed) \end{aligned}$$

$$\begin{aligned} plant \;\triangleq\; & \wedge\ initial.state \\ & \wedge\ \Box[step \vee new.event]_w \\ & \wedge\ WF_w(step) \\ & \wedge \bigwedge_{1 \le i \le n} WF_w(close.i) \end{aligned}$$

Up to now only states of the statechart have been used in the specification, but it is also possible to refer to events in the formulae. E.g. requirement 3 which expresses, that all requests are served, can be described by

$$\mathsf{press\_}i \in E_{exists} \;\leadsto\; in(\mathsf{floor\_}i\mathsf{.door.opened})$$

That is, if the event press_i is generated, there is eventually a state, in which the door is opened on the corresponding floor. Using this formulation, the states floor_i.request are not needed to express this requirement, but it is still possible to use them to specify other requirements or to describe whether there is a request, which has not been served yet.

6 Controlling the lift

In this section the lift example is continued by describing two slightly different versions of a controller behaving according to the specification. In the next section it will be proven formally that both versions actually satisfy the requirements, thereby assuming a property of the environment, viz. the lift is not hold fast on a floor. To describe the controllers the formalism of statecharts without extensions in temporal logic is used.

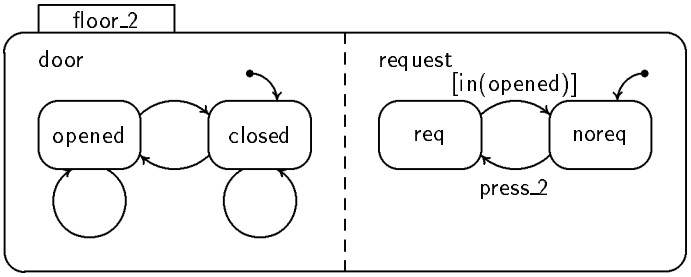


Figure 16: Lift example, variation of the plant.

To effectively close the control loop of the plant via a controller, this must be able to sense the status of the plant and to send stimuli to it. This information has been missing in the description of the from a users point of view (figure 10). This view is extended by the events used to control the plant as shown in figure 17. The events **up** and **down** are used to command the lift to move to the next floor and the events **open_i** to open the doors on the corresponding floors. Remember, that the doors are closed automatically by the plant, no stimuli of the control is necessary to do this. Because the view of the plant of the controller has been considered as a restriction of the the view of an user, the self loops in this statechart have to be restricted, too. The conditions of these transitions express explicitly, that the lift cannot stay idle, when it is commanded by the controller to do something. After hiding the additional events used to communicate with the controller it can be shown, that this description is also a refinement of the original description of the plant.

The two versions of the controller differ in how they sense the status of the plant. The first version directly uses the states in the statechart describing the plant to check whether there are pending requests. This implies, that it does not need to store this information by itself. The second version uses events generated by the plant indicating that a request has been served, therefore it has to store by itself, which requests are still pending. In the second version the the plant and the controller are only communicating by exchanging events.

To combine controller and plant there are two possibilities in the language statecharts. At first, both can be seen as independent activities exchanging information. But therefore it would be necessary to introduce the semantics of activity charts, leading to unnecessary and distracting technical detail here. At second, plant and controller can be seen as two orthogonal *and*-states. Steps in plant and controller are then taken synchronously and exchange of information is accomplished by the broadcasting of events and by refering to states in conditions.

At last, it has to be ensured, that only the controller is able to generate the stimuli necessary to control the plant. Therefore, the TLA-action *new.event* (14) will be restricted appropriately,

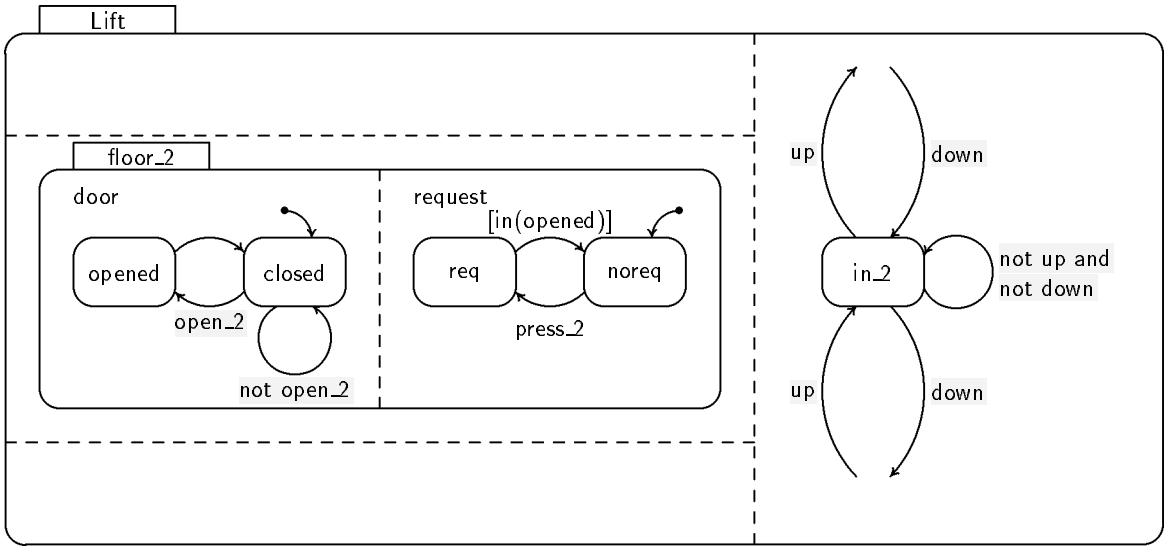


Figure 17: Stimuli of the plant

viz. only requests can be generated by the environment. Let the set *external.events* denote the events $\{\mathsf{press\_}i \mid 1 \leq i \leq n\}$, then *new.event* can be redefined as:

$$
\begin{array}{l}
new.event: \\
new.event \triangleq \begin{array}{l}
\wedge\ E_{exists} \subsetneq E'_{exists} \\
\wedge\ E'_{exists} \setminus E_{exists} \subseteq external.events \\
\wedge\ S'_{active} = S_{active} \\
\wedge\ compound' = \emptyset
\end{array}
\end{array}
$$

6.1 Controller sensing states

The task to serve a request can be decomposed into two subtasks: Move the lift cabin to a requested floor, then open the door to serve the request actually. The second subtask will be considered first.

As in the description of the plant, there is a state in the controller for each floor in the building. Each of these states has three substates indicating whether the door is opening, it is opened or the controller is ready to move to another floor or to serve a request. These states are used to serve a request if the lift is on a requested floor. Some of these states and some of the corresponding transitions are shown in the statechart in figure 18. The events open_*i* are generated, when the controller is starting to serve a request. Control returns to the state floor_*i*.in when the door in the plant is closed again. The states opening are necessary to introduce a delay of the controller to synchronize it with the plant. Otherwise, when serving a request, one would enter a configuration, in which the states lift.floor_*i*.door.closed and control_1.floor_*i*.opened are active and the event open_*i* exists. In the next step, the door would open, but because it is still closed at the beginning of it, the condition closed*i* would be true and the transition from the state control_1.floor_*i*.opened to control_1.floor_*i*.in can be taken. But as the condition req*i* is still true in this configuration, the controller would start a second time to serve the request.

When changing from one of the states floor_*i*.in to floor_*j*.in $(i \neq j)$ one of the events up or down is generated. This implies, that when moving the lift, the states floor_*i*.in of the controller and lift.in_*j* of the plant will be off by one. Moving the lift is restricted by the control strategy described in the specification: The lift may change its direction only if there is no further request in the same direction. This strategy has been described in part by the statechart in figure 13, which becomes a part of the controller. Using this chart, control may change from a state floor_*i*.in to a state floor_*j*.in if it is in the appropriate substate of direction. Furthermore, these transitions are restricted by the requirement, that the lift cannot leave a floor with an unserved request. This implies, that in the controller one of the states floor_*i* must not be left if the plant recognizes a request in the current step. This happens if the appropriate event press_*i* is in the set E_{exists}. Hence, these transitions are also restricted by the absence of the external events press_*i*.

The following informal arguments can be used to show, that the combination of plant and controller satisfies the requirements stated in section 5.2.

1. The events up resp. down are generated only when changing from state floor_*i*.in to floor_*j*.in. But in state floor_*j*.in all doors are closed and remain closed by taking such a transition, hence all doors are closed when the lift is moving.

2. By the condition [not reqi] and the negated event not press_*i* it is guaranteed, that no floor with a pending request is left.

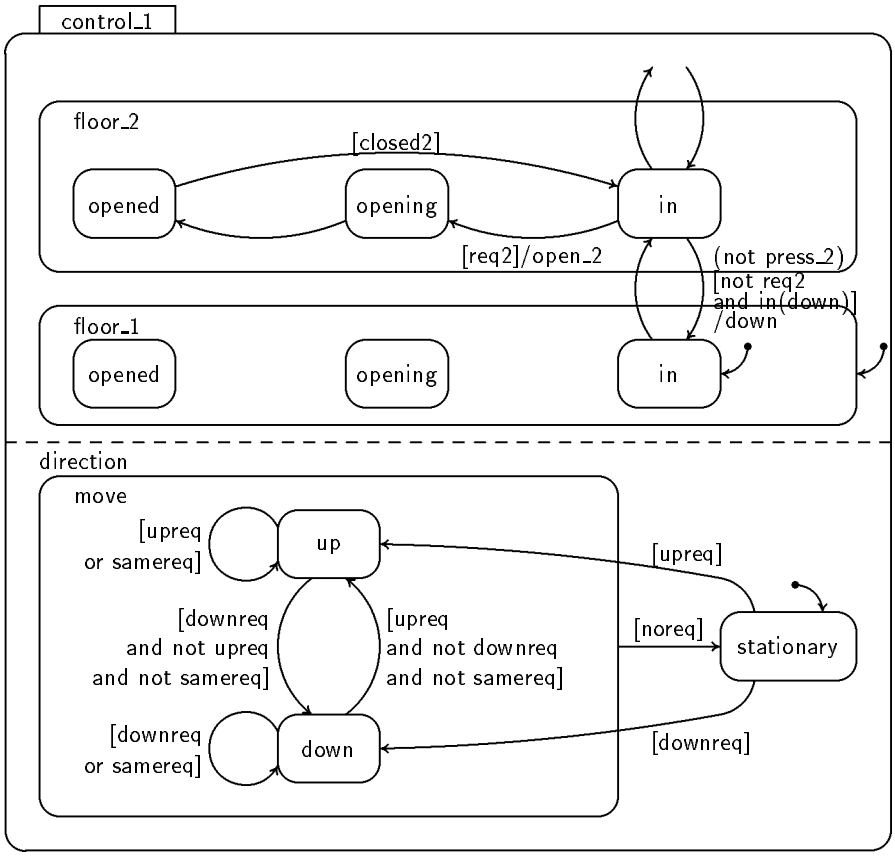
3. As long as there is a request, control is in one of the states move.up resp. move.down and this keeps the lift moving.[25]

4. The condition [reqi] guarantees, that a door is opened only if there is a request.

5. Realized by reusing the part of the specification which describes the control strategy concerning the direction of movement.

6. In the controller it is recognized, that a request has been served as soon as the appropriate door opens. If that was the last pending request, the controller is in the state stationary before it leaves the state floor_*i*.opened and therefore, no further events up or down are generated.

[25] It will be shown in the formal proofs that it is possible to hold fast the lift on a certain floor, contradicting the specification.



req2 is defined as in(lift.floor_2.request.req)
closed2 is defined as in(lift.floor_2.door.closed)
upreq, downreq, etc. are defined as in section 5.2.4
The transitions in the states floor_*i* are similar to those of floor_2.
The labels of some of the transitions between the states floor_*i* are omitted.

Figure 18: First version of the controller

7. See 3 above.

These informal arguments will be proven formally in section 7.

6.2 Controller using events

The second version of the controller does not sense directly the state of the plant. Instead, plant and controller are communicating only by exchanging events. The events **up**, **down** and **open_*i*** are used as before to control the plant. To recognize, whether there is a request, the controller itself uses the events **press_*i***. In addition, the plant generates the appropriate one of the events **served_*i***, when a request has been served, see figure 19.

This version of the controller stores by itself which of the floors are requested. For each of the floors, there are two states **req_*i*** and **noreq_*i***, respectively. The transition from **noreq_*i*** to **req_*i*** is triggered by the event **press_*i***. The state **req_*i*** is left, when the event **served_*i*** occurs. According to the existence of the event **press_*i*** it is reentered or the state **noreq_*i*** is entered, see figure 20.

Serving requests can now be controlled in the same way as in the first version of the controller by using these states instead of the corresponding states of the plant. Because the plant explicitly generates an event when a request has been served, the delay states **opening** of the first version are not necessary here. The movement of the lift can also be controlled as in the first version, again using the extension of the plant to describe the control strategy. The definitions of the conditions **upreq**, **downreq**, etc. have to be adapted to refer to the states **req_*i*** and **noreq_*i*** of the controller instead of the states of the plant.

Essentially the same arguments as for the first version can be applied to show correctness of the second version of the controller. The differences in the proofs are shown in section 7.4.

7 Proving correctness

In this section it will be proven formally, that both versions of the controller are correct with respect to the specification. Thereby it is shown, that descriptions in the combined language of statecharts and temporal logic are amenable to formal analysis. The proofs have been

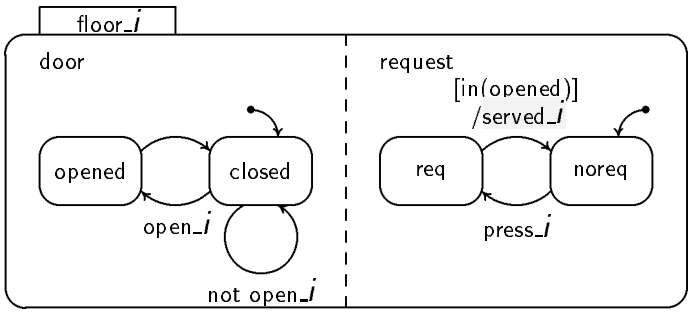


Figure 19: Events generated by the plant

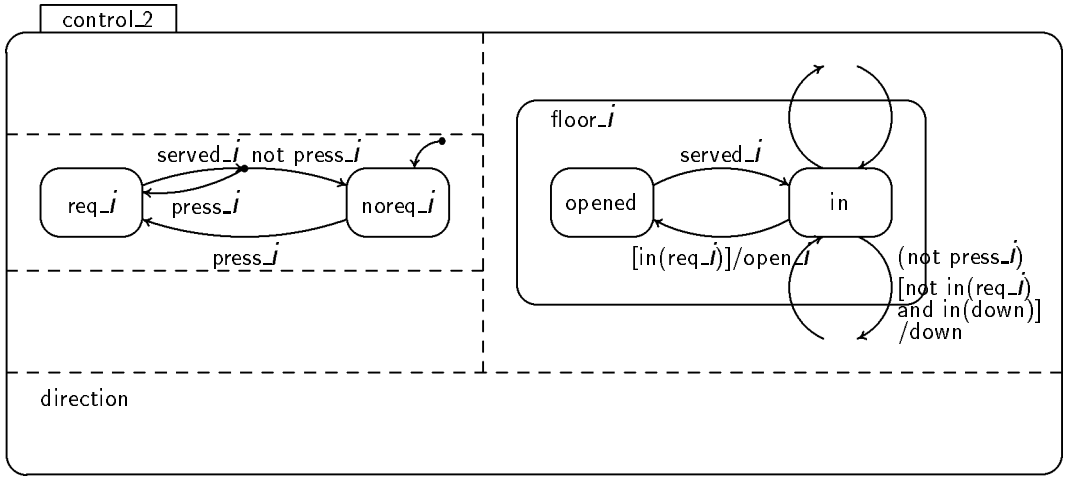
done without tool support, and hence, as one might expect, they have been quite tedious. Nevertheless a flaw of the controller has been detected throughout the proof attempt: It is possible to hold fast the lift on a floor and consequently the controller is not able to command the lift to serve each request. Hence, correctness of the controller with respect to its specification is relative to an assumption about the environment which guarantees, that this situation can not occur.

The proof itself is a refinement proof similar to that presented in [Lamport, 1994b]. It is shown, that the combination of the plant and a controller – the actual closed-loop behaviors – is a refinement of the specification – the desired closed-loop behaviors –, whereby each of these descriptions restricted appropriately to the externally visible states and events. Reasoning about statecharts in the proofs is facilitated by some general lemmata about transitions which can be taken independent of other ones.

At first the correctness proof of the first version of the controller is given in full detail. Therefore the refinement mappings have to be defined, thereby creating several independent proof obligations. Before discharging these obligations, the proofs of the general lemmata about transitions are given. Afterwards some comments on the proof are given. The section finishes with the parts of the proof of the second version of the controller, which are different from the first one.

7.1 Defining the refinement

When comparing the controlled plant and its specification, only the externally visible behavior of the controlled plant is of interest. Similar to the definition of *lift.final* (21) the flexible variables E_{obs} and S_{obs} are used to denote the sets of visible events and states, respectively. Let *plant* || *controller* denote the temporal semantics of the controlled plant. This is described by a statechart consisting of an *and*-state with the statecharts describing the plant, figure 10, and one of the versions of the controller, figures 18 or 20, as direct descendants of it.



The substate direction is the same as in figure 18.
The labels of some of the transitions are omitted.

Figure 20: Second version of the controller

$$
\begin{array}{l}
\mathit{controlled.plant}: \\
\mathit{controlled.plant} \;\triangleq\; \exists\, S_{\mathit{active}}, E_{\mathit{exists}} : \wedge\ \mathit{plant} \parallel \mathit{controller} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \Box S_{\mathit{obs}} = S_{\mathit{active}} \cap S_{\mathit{plant}} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \Box E_{\mathit{obs}} = E_{\mathit{exists}} \cap \mathit{external.events}
\end{array}
$$

Proving correctness of a controller means to prove validity of the formula

$$
\mathit{controlled.plant} \;\Rightarrow\; \mathit{lift.final} \tag{22}
$$

which states that the controlled plant is a refinement of the specification.

When expanding this formula, the hidden variables of the specification and the controlled plant which have the same names in the temporal semantics, have to be distinguished. Therefore the superscripts ^{cp} (*c*ontrolled *p*lant) and ^{sp} (*sp*ecification) will be used. These superscripts will be used for formulae too, meaning that the variables with the same superscript are refered to by the formula. The externally visible variables (S_{obs}, E_{obs}, S_{plant}, and $\mathit{external.events}$) need not be distinguished, more even, they should be the same for both specification and controlled plant to relate them. The expanded goal to prove is

$$
\begin{array}{l}
\exists\, S^{cp}_{\mathit{active}}, E^{cp}_{\mathit{exists}} : \\
\wedge\ \mathit{initial.state}^{cp} \\
\wedge\ \Box \left[\begin{array}{l} \vee\ \mathit{step}^{cp} \\ \vee\ \mathit{new.event}^{cp} \end{array} \right]_{w^{cp}} \\
\wedge\ \mathit{WF}_{w^{cp}}(\mathit{step}^{cp}) \\
\wedge\ \Box S_{\mathit{obs}} = S^{cp}_{\mathit{active}} \cap S_{\mathit{plant}} \\
\wedge\ \Box E_{\mathit{obs}} = E^{cp}_{\mathit{exists}} \cap \mathit{external.events}
\end{array}
\quad \Rightarrow \quad
\begin{array}{l}
\exists\, S^{sp}_{\mathit{active}}, E^{sp}_{\mathit{exists}} : \\
\wedge\ \mathit{initial.state}^{sp} \\
\wedge\ \Box \left[\begin{array}{l} \vee \wedge\ \mathit{step}^{sp} \\ \quad \wedge\ \mathit{staying}^{sp} \\ \vee\ \mathit{new.event}^{sp} \end{array} \right]_{w^{sp}} \\
\wedge\ \Box \mathit{safe.door}^{sp} \\
\wedge \wedge\ \mathit{WF}_{w^{sp}}(\mathit{step}^{sp}) \\
\quad \wedge \bigwedge_{1 \le i \le n} \mathit{WF}_{w^{sp}}(\mathit{grant.i}^{sp}) \\
\wedge\ \Box S_{\mathit{obs}} = S^{sp}_{\mathit{active}} \cap S_{\mathit{plant}} \\
\wedge\ \Box E_{\mathit{obs}} = E^{sp}_{\mathit{exists}} \cap \mathit{external.events}
\end{array}
\tag{23}
$$

To prove this formula it is necessary to remove the existential quantifiers. Following the example in [Lamport, 1994b], a state function $\overline{S^{sp}_{\mathit{active}}}$, which describes how the controlled plant realizes the 'abstract' variable S^{sp}_{active}, and similar functions $\overline{E^{sp}_{\mathit{exists}}}$ and $\overline{\mathit{compound}^{sp}}$, are defined. These state functions are refinement mappings in the sense of [Abadi and Lamport, 1991].

As the set of states in the specification is a subset of the states of the controlled plant, $\overline{S^{sp}_{\mathit{active}}}$ defines a restriction of S^{cp}_{active} to the states of the specification. Hence, similar states, i.e. the state direction and its substates, are identified. Let $S^{cp}_{\mathit{controller}}$ denote the set of states consisting of control_1[26] and its substates except direction (and its substates). One has

$$
\overline{S^{sp}_{\mathit{active}}} \;\triangleq\; S^{cp}_{\mathit{active}} \setminus S^{cp}_{\mathit{controller}} \tag{24}
$$

Similarly, let $\mathit{internal.events}^{cp}$ and $T^{cp}_{\mathit{controller}}$ denote the internal events of the controlled plant and the transitions between the states $S^{cp}_{\mathit{controller}}$, respectively.

$$
\overline{E^{sp}_{\mathit{exists}}} \;\triangleq\; E^{cp}_{\mathit{exists}} \setminus \mathit{internal.events}^{cp} \tag{25}
$$

[26] In the proof of the second version of the controller, the state control_2 and its substates, again except direction (and its substates), have to be taken.

$$\overline{compound^{sp}} \triangleq compound^{cp} \setminus T^{cp}_{controller} \tag{26}$$

Using the proof rules E1 and E2 the existential quantifiers can be removed from the expanded proof goal (23). The proof rule E2 is similar to $\exists$-introduction, as the quantified variables do not occur free in the conclusion. The proof rule E1 eliminates the existential quantifiers in the conclusion by skolemization with the refinement mappings. Let $\overline{F}$ denote, for any formula F, the formula $F(\overline{S^{sp}_{active}}/S^{sp}_{active}, \overline{E^{sp}_{exists}}/E^{sp}_{exists}, \overline{compound^{sp}}/compound^{sp})$. Thereby $F(a/b)$ means substitution of a for b in F. Removing the existential quantifiers then gives

$$\begin{array}{l} \wedge\ initial.state^{cp} \\ \wedge\ \Box \left[\begin{array}{l} \vee\ step^{cp} \\ \vee\ new.event^{cp} \end{array} \right]_{w^{cp}} \\ \wedge\ WF_{w^{cp}}(step^{cp}) \\ \wedge\ \Box S_{obs} = S^{cp}_{active} \cap S_{plant} \\ \wedge\ \Box E_{obs} = E^{cp}_{exists} \cap external.events \end{array} \Rightarrow \begin{array}{l} \wedge\ \overline{initial.state^{sp}} \\ \wedge\ \Box \left[\begin{array}{l} \vee \wedge\ \overline{step^{sp}} \\ \quad \wedge\ \overline{staying^{sp}} \\ \vee\ \overline{new.event^{sp}} \end{array} \right]_{\overline{w^{sp}}} \\ \wedge\ \Box \overline{safe.door^{sp}} \\ \wedge \wedge\ \overline{WF_{w^{sp}}(step^{sp})} \\ \quad \wedge \bigwedge_{1 \le i \le n} \overline{WF_{w^{sp}}(grant.i^{sp})} \\ \wedge\ \Box S_{obs} = \overline{S^{sp}_{active}} \cap S_{plant} \\ \wedge\ \Box E_{obs} = \overline{E^{sp}_{exists}} \cap external.events \end{array} \tag{27}$$

The conjuncts in the conclusion of the formula above can be proven one by one. The safety property $\Box safe.door$ will be proven separately of the refinement of the action *step*, but it would also be possible to rewrite the formulation of it, see the formulation of the safety property (18). The following goals establish validity of (27).

$$initial.state^{cp} \quad \Rightarrow \quad \overline{initial.state^{sp}} \tag{28}$$

$$\Box S_{obs} = S^{cp}_{active} \cap S_{plant} \quad \Rightarrow \quad \Box S_{obs} = \overline{S^{sp}_{active}} \cap S_{plant} \tag{29}$$

$$\Box E_{obs} = E^{cp}_{exists} \cap external.events \ \Rightarrow \ \Box E_{obs} = \overline{E^{sp}_{exists}} \cap external.events \tag{30}$$

$$\begin{array}{l} \wedge\ initial.state^{cp} \\ \wedge\ \Box \left[\begin{array}{l} \vee\ step^{cp} \\ \vee\ new.event^{cp} \end{array} \right]_{w^{cp}} \end{array} \Rightarrow \ \Box \overline{safe.door^{sp}} \tag{31}$$

$$\Box \left[\begin{array}{l} \vee\ step^{cp} \\ \vee\ new.event^{cp} \end{array} \right]_{w^{cp}} \Rightarrow \Box \left[\begin{array}{l} \vee \wedge\ \overline{step^{sp}} \\ \quad \wedge\ \overline{staying^{sp}} \\ \vee\ \overline{new.event^{sp}} \end{array} \right]_{\overline{w^{sp}}} \tag{32}$$

$$\begin{array}{l} \wedge\ initial.state^{cp} \\ \wedge\ \Box \left[\begin{array}{l} \vee\ step^{cp} \\ \vee\ new.event^{cp} \end{array} \right]_{w^{cp}} \\ \wedge\ WF_{w^{cp}}(step^{cp}) \end{array} \Rightarrow \ \overline{WF_{w^{sp}}(step^{sp})} \tag{33}$$

$$\begin{array}{l} \wedge\ initial.state^{cp} \\ \wedge\ \Box \left[\begin{array}{l} \vee\ step^{cp} \\ \vee\ new.event^{cp} \end{array} \right]_{w^{cp}} \\ \wedge\ WF_{w^{cp}}(step^{cp}) \end{array} \Rightarrow \ \bigwedge_{1 \le i \le n} \overline{WF_{w^{sp}}(grant.i^{sp})} \tag{34}$$

7.2 Useful lemmata

The following lemmata are useful to reason about statecharts in TLA. They are the formal counterpart for most of the informal reasoning following in the proofs. Most of the time they will be used without explicit reference.

A compound transition is *independent* of other transitions if there is no other compound transition with a higher or the same scope which is simultaneously enabled.

$$
\begin{array}{l}
independent.transition : 2^T \\
independent.transition(T_1) \triangleq compound.enabled(T_1) \\
\qquad \Rightarrow \bigwedge_{T_2 \in 2^T} \begin{array}{l} \wedge\ one.compound(T_2) \\ \wedge\ substate^*(scope(T_2), scope(T_1)) \\ \wedge\ T_1 \neq T_2 \\ \Rightarrow \neg compound.enabled(T_2) \end{array}
\end{array}
$$

One can see, that in the statecharts describing the controlled plant, see the figures 10, 17, and 18, most of the transitions are independent. Exceptions are the transitions from the state stationary to the states move.up and move.down, respectively. These can be enabled simultaneously, expressing some kind of allowed nondeterminism. Further exceptions are the transitions between different states in_*i* of the lift. This is not a desired situation, but in the following it will be shown, that at most one of the events up and down can be existing in a configuration. Hence, these transitions are independent with respect to the temporal semantics of the controlled plant. The transitions between different states floor_*i*.in of the controller are actually independent, because the states move.up and move.down cannot be active simultaneously.

An important property of independent transitions is, that if one is enabled, then it is executable, too. (Assuming, that it is a well formed statechart)

Lemma 7.1

$$
well.formed \Rightarrow \bigwedge_{T_1 \in 2^T} \begin{array}{l} \wedge\ independent.transition(T_1) \\ \wedge\ compound.enabled(T_1) \\ \Rightarrow compound.executable(T_1) \end{array}
$$

Proof: Follows directly from the definitions of *compound.enabled* and *independent.transition*. ■

This implies, that if an independent transition is enabled, it will be taken in the next *step*-step.

Lemma 7.2

$$
well.formed \Rightarrow \bigwedge_{T_1 \in 2^T} \begin{array}{l} \wedge\ independent.transition(T_1) \\ \wedge\ compound.enabled(T_1) \\ \wedge\ step \\ \Rightarrow T_1 \in compound' \end{array}
$$

Proof: By the lemma above, the transition T_1 is executable, too. Because it is an independent transition, there is no other enabled transition with the same or a higher scope. This implies, that the scope of T_1 is orthogonal to all other transitions executable in the current configuration. Hence, it is a member of each maximal set of executable transitions. The definitions of *maximal.step* and *step* then imply $T_1 \in compound'$. ∎

Not only can independent transitions be executed, but eventually this will happen, if they are enabled continuously. This means, that independent transitions are weakly fair.

Lemma 7.3

$$well.formed \Rightarrow \bigwedge_{T_1 \in 2^T} independent.transition(T_1) \Rightarrow WF_w(step \wedge take.step(\{T_1\}))$$

Proof: Whenever the transition T_1 is enabled, then the TLA-action $step \wedge take.step(\{T_1\})$ is enabled, too. The TLA-action *step* is weakly fair, hence it will be taken eventually. By lemma 7.2, T_1 will be taken in this step. ∎

Remember that a well formed statechart remains well formed during its execution, because its temporal semantics leaves the underlying sets of states and transitions unchanged. Hence these lemmata can be applied in each state of the temporal semantics.

7.3 Proof of correctness for controller version 1

At first the implications concerning the initial states and the sets of observable events and states are proven. Then, some invariants, which are useful for the proofs of the safety properties, are proven. At last it is proven, that the the controlled plant is fair and that each request is served, under some assumptions about the environment. The overall structure of the proof, that is, which property contributes to the proof of which other one, is shown at the end of this section in figure 21.

7.3.1 The easy parts of the proof

To prove the implications concerning the initial state and the sets of observable states and events, it is sufficient to expand the definitions of the formulae and to apply a single temporal proof rule for the latter implications.

$$\begin{array}{lll}
initial.state^{cp} & \Rightarrow & \overline{initial.state^{sp}} \\
\equiv \ \wedge\ S^{cp}_{active} = \emptyset & \Rightarrow & \wedge\ \overline{S^{sp}_{active}} = \emptyset \\
\quad \wedge\ E^{cp}_{exists} = \emptyset & & \wedge\ \overline{E^{sp}_{exists}} = \emptyset \\
\quad \wedge\ compound^{cp} = \emptyset & & \wedge\ \overline{compound^{sp}} = \emptyset \\
\equiv \ \wedge\ S^{cp}_{active} = \emptyset & \Rightarrow & \wedge\ S^{cp}_{active} \setminus S^{cp}_{controller} = \emptyset \\
\quad \wedge\ E^{cp}_{exists} = \emptyset & & \wedge\ E^{cp}_{exists} \setminus internal.events^{cp} = \emptyset \\
\quad \wedge\ compound^{cp} = \emptyset & & \wedge\ compound^{cp} \setminus T^{cp}_{controller} = \emptyset
\end{array}$$

The last implication is trivially valid, hence (28) is valid too.

$$
\begin{array}{rlll}
 & S_{obs} = S^{cp}_{active} \cap S_{plant} & \Rightarrow & S_{obs} = \overline{S^{sp}_{active}} \cap S_{plant} \\
\equiv & S_{obs} = S^{cp}_{active} \cap S_{plant} & \Rightarrow & S_{obs} = (S^{cp}_{active} \setminus S^{cp}_{controller}) \cap S_{plant}
\end{array}
$$

The definition of $S^{cp}_{controller}$ implies $S^{cp}_{controller} \cap S_{plant} = \emptyset$, hence the implication above is valid. Using the proof rule STL4 (29) can be concluded. The proof of (30) is similar.

7.3.2 Invariants used in the proof

Before showing the other proof goals, some useful invariants of the controlled plant are shown. As only the controlled plant is considered here, but not the specification, the superscript ^{cp} is omitted in this section. The invariants can be classified as follows:

- Concerning the movement of the lift
- Concerning the doors
- Concerning requests and the direction of movement

In the following, the position of the lift is often refered to. Therefore a function *lift.floor* and a similar function *control.floor* are defined to refer to these positions. In the initial state no state of the statechart is active. To make the functions applicable to all states we define this value to be the one after taking the initial transition.

$$
\begin{array}{l}
lift.floor : \rightarrow \{1 \ldots n\} \\
lift.floor = i \ \triangleq \ \vee\ in(\mathsf{lift.in\_}i) \\
\qquad\qquad\qquad\quad \vee \wedge\ i = 1 \\
\qquad\qquad\qquad\qquad \wedge\ S_{active} = \emptyset
\end{array}
$$

$$
\begin{array}{l}
control.floor : \rightarrow \{1 \ldots n\} \\
control.floor = i \ \triangleq \ \vee\ in(\mathsf{control\_1.floor\_}i) \\
\qquad\qquad\qquad\quad \vee \wedge\ i = 1 \\
\qquad\qquad\qquad\qquad \wedge\ S_{active} = \emptyset
\end{array}
$$

Invariants concerning the movement of the lift: As already mentioned, when the lift is moving, the positions of the lift and of the controller are off by one. In this situation, the appropriate event (up resp. down) is generated.

$$
\begin{array}{l}
floor.difference : \\
floor.difference \ \triangleq \ \wedge\ |lift.floor - control\_floor| \leq 1 \\
\qquad\qquad\qquad\quad \wedge\ lift.floor < control\_floor \Leftrightarrow \mathsf{up} \in E_{exists} \\
\qquad\qquad\qquad\quad \wedge\ lift.floor > control\_floor \Leftrightarrow \mathsf{down} \in E_{exists}
\end{array}
$$

The invariant itself is

$$
\Box \quad floor.difference \tag{35}
$$

Proof : It is shown that this invariant is implied by the temporal semantics of the controlled plant. Therefore it has to be shown, that it is valid in the initial state of the controlled plant and that it cannot be falsified by a step of the controlled plant.

The proof rule INV1 can be applied to the latter proposition, yielding

$$(\mathit{floor.difference} \wedge \Box[\mathit{step} \vee \mathit{new.event}]_w) \;\Rightarrow\; \Box \mathit{floor.difference}$$

The hypothesis of this implication can be discharged by the temporal semantics of the controlled plant and by showing that *floor.difference* is valid initially.

Expanding the definition of *initial.state* one has $S_{active} = \emptyset$ and $E_{exists} = \emptyset$. This implies that *floor.difference* is initially valid.

To show that *floor.difference* is preserved by a step it has to be shown, that the following is valid

$$(\mathit{floor.difference} \wedge [\mathit{step} \vee \mathit{new.event}]_w) \;\Rightarrow\; \mathit{floor.difference}'$$

After expanding the definitions and applying propositional logic the following three cases have to be considered:

$\mathit{floor.difference} \wedge w = w' \;\Rightarrow\; \mathit{floor.difference}'$: Trivial.

$\mathit{floor.difference} \wedge \mathit{new.event} \;\Rightarrow\; \mathit{floor.difference}'$: Due to the definition of *new.event* activity of states does not change and no internal event can be added to the set E_{exists}. Hence validity is preserved in this case.

$\mathit{floor.difference} \wedge \mathit{step} \;\Rightarrow\; \mathit{floor.difference}'$:
Let us assume first that $\mathit{lift.floor} = \mathit{control.floor}$. This implies $\mathsf{up}, \mathsf{down} \notin E_{exists}$ and thereby[27] $\mathit{lift.floor} = \mathit{lift.floor}'$. The events up and down are generated only if $\mathit{control.floor}' \neq \mathit{control.floor}$. Furthermore $|\mathit{control.floor}' - \mathit{control.floor}| \leq 1$. Hence $\mathit{floor.difference}'$ can be concluded.
Now let us assume $|\mathit{lift.floor} - \mathit{control.floor}| = 1$, without loss of generality $\mathit{lift.floor} < \mathit{control.floor}$. Hence we have $\mathsf{up} \in E_{exists}$ and $\mathit{lift.floor}' = \mathit{lift.floor} + 1$. Arguing as above $\mathit{floor.difference}'$ can be concluded. ∎

Invariants concerning the doors: These invariants relate the various states opened and opening and the events open_*i*.

The first of these invariants states, that the controller tries to open a door only when the lift is on the corresponding floor.

$$\Box \bigwedge_{1 \leq i \leq n} (\,\mathsf{open\_}i \in E_{exists} \;\Rightarrow\; \mathit{lift.floor} = i\,) \tag{36}$$

Proof: The proof is similar to that of (35). The critical case is again taking the TLA-action *step*. It has to be shown

$$\begin{aligned}
&\wedge\, (\,\mathsf{open_}i \in E_{exists} \;\Rightarrow\; \mathit{lift.floor} = i\,)\\
&\wedge\, \mathit{step}\\
&\Rightarrow\, (\,\mathsf{open_}i \in E'_{exists} \;\Rightarrow\; \mathit{lift.floor}' = i\,)
\end{aligned}$$

[27] This and the following properties are direct consequences of the temporal semantics.

By the temporal semantics, open_$i \in E_{exists}$ implies in(floor_i.opening). But then open_$i \notin E'_{exists}$ and the proposition is vacuously valid. Hence, let us assume open_$i \in E'_{exists}$, which implies in(floor_i.in) and in'(floor_i.opening). This further implies up, down $\notin E'_{exists}$. By (35) it can be concluded that $lift.floor' = control.floor' = i$. When the conclusion is falsified by a step, i.e. $lift.floor' \neq i$ and open_$i \in E'_{exists}$, then the first proposition implies $control.floor \neq i$, but the second one implies in(control_1.floor_i.in) and thereby $control.floor = i$, a contradiction. ∎

Note, that there is a general structure of proofs for properties of the form

$$\begin{array}{l} \wedge\ (\ p\ \Rightarrow\ q\ ) \\ \wedge\ step \\ \Rightarrow\ (\ p'\ \Rightarrow\ q'\ ) \end{array}$$

Taking such a step, the values of p and q may change. But it is sufficient to consider the cases $\neg p \wedge p'$ and $q \wedge \neg q'$, i.e. the premise becomes true and the conclusion becomes false, respectively. The cases where the truth values are preserved, the premise becomes false, or the conclusion becomes true, automatically satisfy the formula above. Most of the following proofs will have this structure.

A direct corollary of (36) is, that when control is in one of the states floor_i.opening, then the lift is on the corresponding floor. Similar arguments can be applied to the states floor_i.opened:

$$\begin{array}{l} \Box\ \bigvee_{1 \le i \le n} \vee\ in(\text{control\_1.floor\_}i\text{.opening}) \\ \qquad \vee\ in(\text{control\_1.floor\_}i\text{.opened}) \\ \qquad \Rightarrow\ lift.floor = control.floor \end{array} \tag{37}$$

On the other hand, it can be shown, that control cannot leave one of the states control_1.floor_i.opened, as long as the door on the corresponding floor is opened.

$$\begin{array}{l} \Box\ \bigwedge_{1 \le i \le n} in(\text{lift.floor\_}i\text{.door.opened}) \\ \qquad \Rightarrow in(\text{control\_1.floor\_}i\text{.opened}) \end{array} \tag{38}$$

Proof: Proceeding as above it must be shown for each floor i, that

$$\begin{array}{l} \wedge\ (\ in(\text{lift.floor\_}i\text{.door.opened})\ \Rightarrow\ in(\text{control\_1.floor\_}i\text{.opened})\ ) \\ \wedge\ step \\ \Rightarrow\ (\ in'(\text{lift.floor\_}i\text{.door.opened})\ \Rightarrow\ in'(\text{control\_1.floor\_}i\text{.opened})\ ) \end{array}$$

Let us assume, that in'(lift.floor_i.door.opened) is valid. This implies $\neg in$(lift.floor_i.door.opened) and thereby open_$i \in E_{exists}$. Furthermore in(control_1.floor_i.opening) and in'(control_1.floor_i.opened) are valid.
Steps which falsify the conclusion (in'(control_1.floor_i.opened)) imply in(lift.floor_i.door.closed) and in'(lift.floor_i.door.closed), hence the premise is falsified, too. ∎

Invariants concerning requests and the direction of movement: At first it will be shown, that whenever there is no request, but the controller is ready to move the lift, the state stationary is active. This implies, that the lift will not move in this situation.

$$\Box \wedge \bigvee_{1 \le i \le n} in(\mathsf{control\_1.floor\_}i\mathsf{.in}) \tag{39}$$
$$\wedge\ check.condition.\mathsf{noreq}$$
$$\Rightarrow in(\mathsf{stationary})$$

Proof: Let us consider first the case $in(\mathsf{stationary}) \wedge \neg in'(\mathsf{stationary})$. This implies, that at least one of the conditions upreq or downreq has become valid, i.e. there is a new request for at least one other floor. But $in(\mathsf{stationary})$ implies, that the lift cannot move in this step, hence it is not possible to serve a request on another floor, implying $\neg check.condition.\mathsf{noreq}'$.

Now let us assume, that the premises becomes valid. We will show first, that in this case, the first premise becomes valid while the second one is already valid. When the second premise becomes valid, the definition of noreq implies $in(\mathsf{lift.floor\_}i\mathsf{.request.req}) \wedge in(\mathsf{lift.floor\_}i\mathsf{.request.noreq})$ for one of the floors i. This implies in turn $in(\mathsf{lift.floor\_}i\mathsf{.door.opened})$. By (38) one has $in(\mathsf{control\_1.floor\_}i\mathsf{.opened})$ and thereby $in'(\mathsf{control\_1.floor\_}i\mathsf{.opened})$, contradicting that the first premise is valid after this step. Hence the condition noreq is already valid and remains valid and the first premise becomes valid. Validity of condition noreq is sufficient to conclude $in'(\mathsf{stationary})$ ∎

On the contrary, when the controller is ready to command the lift to move and the state move is active, then actually there is a request.

$$\Box \wedge \bigvee_{1 \le i \le n} in(\mathsf{control\_1.floor\_}i\mathsf{.in}) \tag{40}$$
$$\wedge\ in(\mathsf{move})$$
$$\Rightarrow \neg check.condition.\mathsf{noreq}$$

Proof: This is the contraposition of the invariant above. ∎

These results can be used to relate the state direction and the movement of the lift: The lift does not move, when the state stationary is active.

$$\Box in(\mathsf{stationary}) \tag{41}$$
$$\Rightarrow \mathsf{up}, \mathsf{down} \notin E_{exists}$$

Proof: Falsifying the conclusion means to generate one of these events. But these can only be generated by taking transitions with enabling condition in(up) resp. in(down). This implies further, that the condition noreq is false and thereby $\neg in'(\mathsf{stationary})$. Validating the premise implies that the condition noreq is valid at the beginning of the step. If one of the states control_1.floor_i.in is active, it can be concluded by (39) that the state stationary has already been active, contradicting our assumption. If none of these states is active, it is not possible to generate these events, validating the conclusion. ∎

Similarly it can be stated, that when the lift is commanded to move, the state move is active. It should be easy to see, that the events up and down are corresponding to the appropriate substates of move.

$$\Box\ (\ \mathsf{up}, \mathsf{down} \in E_{exists} \ \Rightarrow\ in(\mathsf{move})\ ) \tag{42}$$

Proof: This invariant is the contraposition of the one above. ∎

Now it is possible to relate requests and the events controlling the movement of the lift.

When the lift is requested to move upwards, then there is actually a request for a floor above the current position of the lift.

$$\Box\,(\,\mathsf{up} \in E_{exists} \;\Rightarrow\; check.condition.\mathsf{upreq}\,) \tag{43}$$

Proof: By (42) $\mathsf{up} \in E_{exists}$ implies $in(\mathsf{move.up})$. Furthermore, one of the states control_1.floor_*i*.in is active, by (40) we conclude $\neg check.condition.\mathsf{noreq}$, which is equivalent to $check.condition.\mathsf{upreq} \wedge check.condition.\mathsf{downreq}$. Because $check.condition.\mathsf{upreq}$ is valid when entering the state move.up and is preserved as long as the controller is in this state, $check.condition.\mathsf{upreq}$ can be concluded. ∎

This invariant (and its dual for moving downwards) can also be stated the other way round. When there is no request, then the lift is not commanded to move

$$\Box\,(\,check.condition.\mathsf{noreq} \;\Rightarrow\; \mathsf{up}, \mathsf{down} \notin E_{exists}\,) \tag{44}$$

When the lift is commanded to move, then there is not only a request for a floor in the corresponding direction, but there is also no request for the floor currently leaving.

$$\Box\,(\,\mathsf{up} \in E_{exists} \;\Rightarrow\; check.condition.\mathsf{not\ samereq}^{28}\,) \tag{45}$$

Proof: Falsifying the conclusion means that the event press_*i* is existing at the beginning of the step, but then no transition leaving the actual floor (in control_1) can be taken, hence $\mathsf{up}, \mathsf{down} \notin E'_{exists}$. Validating the premise implies, that such a transition has been taken. The condition not req*i* implies $in(\mathsf{lift.floor\_}i\mathsf{.request.noreq})$ and the condition not press_*i* preserves this property. Hence $check.condition.\mathsf{not\ samereq}$ is valid, too. ∎

Using the invariants: The use of the invariants in the following proofs is justified by the proof rule INV2. Each step can be strengthened to one which is preserving the invariants. Let *invariant* be the conjunction of the invariants proven so far, then the following equivalence is valid

$$\Box\left[\begin{array}{l}\vee\ step\\ \vee\ new.event\end{array}\right]_w \Leftrightarrow \Box\left[\begin{array}{l}\wedge \vee\ step\\ \quad\vee\ new.event\\ \wedge\ invariant \wedge invariant'\end{array}\right]_w \tag{46}$$

The invariants will be used in the proofs by refering in an informal way to them, but not by introducing them formally in the proofs as shown above.

7.3.3 Proving safety

Using the invariants (38) and (37), it can be shown for each of the floors i

[28] The blank character in not samereq is part of the name of this predicate.

$$
\begin{aligned}
& in^{cp}(\mathsf{lift.floor\_}i\mathsf{.door.opened}) \\
\Rightarrow\ & in^{cp}(\mathsf{control\_1.floor\_}i\mathsf{.opened}) \\
\Rightarrow\ & lift.floor^{cp} = control.floor^{cp}
\end{aligned}
$$

This is actually the invariant $\Box\,(\,in^{cp}(\mathsf{lift.floor\_}i\mathsf{.door.opened}) \;\Rightarrow\; lift.floor^{cp} = i\,)$, which implies $\Box safe.door^{cp}$. The definition of $safe.door$ uses only states in S_{plant}, hence $\Box\overline{safedoor^{sp}}$, i.e. (31), is valid. ■

7.3.4 Proving correctness of the steps

To prove (32) one can get rid of the temporal operators ($\Box$) by applying the proof rule STL4. Expanding the definition of $[\quad]_w$ it remains to show

$$
\begin{array}{lcl}
\vee\ step^{cp} & \Rightarrow & \vee \wedge\ \overline{step^{sp}} \\
\vee\ new.event^{cp} & & \quad \wedge\ \overline{staying^{sp}} \\
\vee\ w^{cp} = w^{cp\prime} & & \vee\ \overline{new.event^{sp}} \\
 & & \vee\ \overline{w^{sp}} = \overline{w^{sp\prime}}
\end{array}
\qquad (47)
$$

This means, that each TLA-step of the controlled plant must also be a valid step in the specification. This is easy to see for stuttering steps and the step $new.event^{cp}$. To show this for $step^{cp}$-steps the individual *and*-states are examined separately. This can be done because there are no transitions between different *and*-states. Except for the state $\mathsf{control\_1.lift}^{cp}$, the *and*-states of the controlled plant are also states of the specification, see the statecharts in figure 18 and figure 15. Therefore properties of this part of the controlled plant are also valid for the specification.

$\mathsf{control\_1.lift}^{cp}$: There is no direct counterpart to this state in the statechart of the specification. But it must be assured, that whenever it is possible to take a transition which implies, that existing events are 'consumed', it is also possible to consume the same events in the statechart of the specification[29]. For each of the states $\mathsf{lift.lift.in\_}i^{sp}$, there is an unrestricted transition to itself, hence it is always possible to take such a step.

$\mathsf{control\_1.direction}^{cp}$: Identical to $\mathsf{lift.direction}^{sp}$.

$\mathsf{lift.floor\_}i\mathsf{.request}^{cp}$: Identical to $\mathsf{lift.floor\_}i\mathsf{.request}^{sp}$.

$\mathsf{lift.floor\_}i\mathsf{.door}^{cp}$: This is a restricted form of $\mathsf{lift.floor\_}i\mathsf{.door}^{sp}$.

$\mathsf{lift.lift}^{cp}$: When a transition from a state $\mathsf{lift.lift.in\_}i^{cp}$ to itself is taken, this can also be taken in the statechart of the specification.

Let us consider, without loss of generality, that the transition from $\mathsf{lift.lift.in\_}i^{cp}$ to $\mathsf{in\_}i{+}1^{cp}$ is taken. It has to be shown, that it is possible to take this transition in the specification statechart, too. Furthermore, it has to be shown, that there is indeed a request above $lift.floor$, otherwise the property *staying* would not have been preserved.

[29] Therefore, there are no steps completely internal to the controller, which could be seen as stuttering steps of the specification. Here, a semantics supporting the synchrony hypothesis to a larger degree would be of help.

The enabling conditions for this transition in the specification are $\mathsf{not\ req}i^{sp}$ and $\mathsf{in(up)}^{sp}$. It must be shown, that these conditions are satisfied. Taking this transition in the controlled plant implies $\mathsf{up} \in E^{cp}_{exists}$, by (42) it follows $\mathsf{in(up)}^{cp}$. The invariant (45) implies, that there is no request for the current position of the lift, i.e. $\mathsf{not\ req}i^{cp}$ is valid.

(43) implies, that there is actually a request for a floor above the current position of the lift. Hence the condition noreq is not valid and $staying^{cp}$ is satisfied. ∎

7.3.5 Proving general fairness

Weak fairness of the $step^{sp}$-steps can be proven by the proof rule WF2 using the following substitutions. Note that the action N has been strengthened explicitly by the invariants.

$$\begin{array}{ll} A = B = step^{cp} & P = F = true \\ M = step^{sp} & N = \wedge\ invariant^{cp} \wedge invariant^{cp\prime} \\ & \quad \wedge \vee\ step^{cp} \\ & \quad\ \ \vee\ new.event^{cp} \\ g = w^{sp} & f = w^{cp} \end{array}$$

which yields

$$\begin{array}{l} \wedge\ \Box \left[\begin{array}{l} \wedge\ invariant^{cp} \wedge invariant^{cp\prime} \\ \wedge \vee\ step^{cp} \\ \quad \vee\ new.event^{cp} \end{array} \right]_{w^{cp}} \\ \wedge\ WF_{w^{cp}}(step^{cp}) \\ \wedge\ \Box true \\ \Rightarrow\ \overline{WF_{w^{sp}}(step^{sp})} \end{array}$$

This implies validity of (33). It remains to show the premises of the proof rule (most of the substitutions above are already performed in these premises):

$$\langle N \wedge step^{cp} \rangle_{w^{cp}} \ \Rightarrow\ \langle \overline{step^{sp}} \rangle_{\overline{w^{sp}}} \tag{48}$$

$$\begin{array}{ll} \wedge\ true \wedge true' & \Rightarrow\ step^{cp} \\ \wedge\ \langle N \wedge step^{cp} \rangle_{w^{cp}} & \\ \wedge\ \overline{Enabled\langle step^{sp} \rangle_{w^{sp}}} & \end{array} \tag{49}$$

$$true \wedge \overline{Enabled\langle step^{sp} \rangle_{w^{sp}}} \ \Rightarrow\ Enabled\langle step^{cp} \rangle_{w^{cp}} \tag{50}$$

$$\begin{array}{ll} \wedge\ \Box[N \wedge \neg step^{cp}]_{w^{cp}} & \Rightarrow\ \Diamond\Box true \\ \wedge\ WF_f(step^{cp}) & \\ \wedge\ \Box true & \\ \wedge\ \Box\Diamond \overline{Enabled\langle step^{sp} \rangle_{w^{sp}}} & \end{array} \tag{51}$$

(51) and (49) are trivial. (50) will be shown first: This premise means, that whenever a $step$-step of the specification is enabled, then one of the controlled plant is enabled too. Expanding the definition of $step$, this simplifies to

$$\overline{\bigvee_{\mathcal{T}_1^{sp} \in 2^{2^{T^{sp}}}} maximal.step(\mathcal{T}_1^{sp})} \ \Rightarrow\ \bigvee_{\mathcal{T}_1^{cp} \in 2^{2^{T^{cp}}}} maximal.step(\mathcal{T}_1^{cp})$$

which means, that whenever it is possible to take a transition in the statechart of the specification, it is also possible to take one in that of the controlled plant. Note, that these two transitions need not be the same ones. As the statechart of the controlled plant is constructed from orthogonal *and*-states which do not interact via transitions crossing the boundary of them, it is sufficient to look at the transitions within the *and*-states independently. To show, that the TLA-action *step* is enabled, it is sufficient to have one *and*-state in the controlled plant, in which it is possible to take a transition in each step. These are the transitions from lift.lift.in_*i* to lift.lift.in_*j* because the triggering events of the outgoing transitions of each of the states lift.lift.in_*i* form a complete case distinction.

Expanding definitions and using the fact, that a *step*-step cannot be a *new.event*-step (due to the variable *compound*), it is sufficient to show the following to conclude validity of the first premise (48).

$$\begin{array}{lll} \wedge\ invariant \wedge invariant' & \Rightarrow & \wedge\ \overline{step^{sp}} \\ \wedge\ step^{cp} & & \wedge\ \overline{w^{sp}} \neq \overline{w^{sp\prime}} \\ \wedge\ w^{cp} \neq w^{cp\prime} & & \end{array}$$

But this has already been shown in the last section when proving the correctness of the *step*-steps. ∎

7.3.6 Each request is served

To prove that each request is granted in the specification, it is sufficient to prove that each one is granted in the controlled plant. Hence only this statechart will be considered and the superscript ^{cp} will be omitted.

Weak fairness is implied by leadsto: $WF_w(grant.i)$ states, that this action cannot be enabled continuously without being taken. *grant.i* has the special property, that it can be disabled only by taking it, i.e.

$$\begin{array}{l} \wedge\ step \\ \wedge\ Enabled\langle grant.i\rangle_w \\ \wedge\ \neg Enabled'\langle grant.i\rangle_w \\ \Rightarrow\ grant.i \end{array}$$

In the specification an alternative formulation of the property to be proven here has been shown. There the following leadsto formula (20) has been used.

$$serve.i \ \triangleq\ in(\mathsf{floor\_}i\mathsf{.reqest.req}) \leadsto in(\mathsf{floor\_}i\mathsf{.door.opened})$$

Because $Enabled\langle grant.i\rangle_w$ implies $in(\mathsf{floor\_}i\mathsf{.request.req})$ and $in(\mathsf{floor\_}i\mathsf{.door.opened})$ implies $\neg Enabled\langle grant.i\rangle_w$ one has

$$serve.i \ \Rightarrow\ WF_w(grant.i) \tag{52}$$

To show that *serve.i* is valid for the controlled plant it will be shown first, that the lift can move in the requested direction, second, that requested floors are reached by the lift and third, if the lift is on a requested floor, the request is eventually served. Transitivity of $\leadsto$ implies (52).

The controller does not fulfill the specification: Throughout the proof attempt it has been discovered, that the controller actually does not satisfy the specification. As easily can be seen, the controller can be hold fast on a floor. Due to the triggering events of the transitions between the states control_1.floor_i, especially not press_i, it is not possible to leave such a state if these events are generated before each step. But in the specification it is possible to move the lift in this situation because one of the states noreq is entered before req is entered again. Hence there is a state, in which the lift is able to move.

One cannot get rid of these events in the controlled plant, because they are necessary to proof the correctness of steps (32). More precisely, they are used in the proof of the invariant (45). The problem of the controller is, that it needs more steps to recognize that a request has been served and to be ready to move again than the lift in the specification[30]. Therefore the controller is to late to use this single state, where moving the lift would be allowed.

Instead of changing the specification or the controller, an assumption that the lift would not be hold fast on a floor by the environment, is defined and used to show correctness relative to this assumption. Let the TLA-action *move* denote the change of control in the controller.

$$
\begin{array}{l}
move: \\
move \triangleq \ \wedge \ step \\
\qquad\quad \wedge \ control.floor \neq control.floor'
\end{array}
$$

Then the assumption can be expressed formally by requiring that this action will be enabled eventually when there is a request on another floor and by requiring that it will be taken eventually.

$$
\begin{array}{l}
\wedge \ \vee \ check.condition.\mathsf{upreq} \\
\quad \vee \ check.condition.\mathsf{downreq} \\
\quad \leadsto \ Enabled\langle move\rangle_w \\
\wedge \ SF_w(move)
\end{array}
\tag{53}
$$

The lift advances to the next floor: It will be shown, that if the lift is requested in the direction corresponding to the active substate of move, then it will move eventually in this direction. This will be shown first for the position of the controller and then for the position of the lift.

$$
\begin{array}{lcl}
\wedge \ check.condition.\mathsf{upreq} & \leadsto & \wedge \ control.floor = i + 1 \\
\wedge \ control.floor = i & & \wedge \ in(\mathsf{up}) \\
\wedge \ in(\mathsf{up}) & &
\end{array}
\tag{54}
$$

Proof: Using the assumption (53) and the proofrule SF1, it can be shown, that the controller moves eventually. The premises $in(\mathsf{up})$ and $control.floor = i$ imply that it moves upward, generating the event up. ∎

Weak fairness of *step* implies $(\ in(\mathsf{up}) \wedge control.floor = j\ ) \leadsto lift.floor = j$ (using the proofrule WF1). By transitivity of $\leadsto$ it can be concluded that the lift itself moves in the right direction eventually. ∎

[30] This is a further indication, that the synchrony hypothesis should be maintained in the temporal semantics to a larger amount than it has been done here.

The lift reaches a requested floor: The proof rule LATTICE is used to show, that a requested floor (in the current direction of the lift) is eventually reached. The substitutions used are

$$
\begin{array}{lcl@{\qquad}lcl}
S & = & \mathbf{N} & \succ & = & > \\
F & = & \wedge\ \mathit{controlled.plant} & G & = & \wedge\ \mathit{in}(\mathsf{lift.floor\_}i\mathsf{.request.req}) \\
 & & \wedge\ \wedge\ \vee\ \mathit{check.condition}.\mathsf{upreq} & & & \wedge\ \mathit{lift.floor} = i \\
 & & \quad \vee\ \mathit{check.condition}.\mathsf{downreq} & & & \\
 & & \quad \rightsquigarrow \mathit{Enabled}\langle\, \mathit{move}\,\rangle_w & & & \\
 & & \wedge\ \mathit{SF}_w(\mathit{move}) & & & \\
H_{\mathit{dist}} & = & \wedge\ \mathit{in}(\mathsf{lift.floor\_}i\mathsf{.request.req}) & & & \\
 & & \wedge\ \mathit{in}(\mathsf{up}) & & & \\
 & & \wedge\ \mathit{lift.floor} \leq i & & & \\
 & & \wedge\ \mathit{dist} = \mathit{lift.floor} - i & & &
\end{array}
$$

The premise of the rule states, that eventually G or H_{dist} with a strictly smaller distance *dist* becomes valid. (54) implies validity of the premise. ∎

The lift moves in the right direction: The same kind of argumentation can be used to show that eventually a state is reached in which there is no further request downward, but the state down is still active. In this situation there can be another request downwards or in the next step the direction changes to up. Because it is not possible to travel infinitely in one direction, the state up will be reached eventually. ∎

The request is served: It remains to show, that when the lift is on a requested floor, the request will be served. Lemma 7.3 and transitivity of $\rightsquigarrow$ imply $\mathit{in}(\mathsf{control\_1.floor\_}i\mathsf{.in}) \rightsquigarrow (\ \mathit{in}(\mathsf{control\_1.floor\_}i\mathsf{.opening}) \wedge \mathsf{open\_}i \in E_{\mathit{exists}}\ ) \rightsquigarrow \mathit{in}(\mathsf{lift.floor\_}i\mathsf{.door.opened})$. Hence *serve.i* (20) and as a consequence the liveness property (52) can be concluded. ∎

7.3.7 Conclusions from this proof

Some observations about the structure of the proof can be given. A first part, quite routine part, of the proof consisted of splitting up the goal into several independent subgoals, thereby following the example of a refinement proof in [Lamport, 1994b]. Because essentially the open- and the closed-loop behaviors of the plant have been compared in the proof, the refinement mappings used here have been restrictions of the sets of states and events of the plant together with the controller to the sets of observable states and events of the plant alone. Hence, the refinement mappings are very simple here. But it is expected, that when the controller itself is subject to refinement, the refinement mappings will be more complicated.

Showing implication of the conditions on initial states and relating the sets of observable states and events was also easy.

The other subgoals can be divided into two subgroups: safety and liveness. The safety proofs follow a pattern which can be described by subsequent strengthening of invariants. Starting points of this reasoning are the temporal semantics and the lemmata on independent transitions, especially lemma 7.2. The intermediate steps connect the inputs with the internal

state of the controlled plant and vice versa this internal state with the observable behavior of the plant. The difficult part here was to find the appropriate invariants from which the safety properties could be concluded in a simple way. It has also been difficult to proof the chains of invariants without introducing cyclic reasoning.

These chains of invariants and the other parts of the proof are depicted in figure 21. The arrows indicate which invariant or property is used in the proof of another one.

The liveness properties are more independent of the invariants, the temporal semantics and the lemmata 7.2 and 7.3 have been used more directly. But it should be noted that one of the safety properties (correctness of steps) has been used in the proof of the liveness properties. Also, a leadsto formulae has been proved instead of a weak fairness property. This made the proof easier, because the proof rules concerning leadsto have simpler premises than those concerning fairness. At second, leadsto is transitive, which makes it easy to partition a proof into one describing a finite sequence of states, one leading to another one.

7.4 Proof of correctness for controller version 2

The proof of this version of the controller is very similar to that of the first one. It starts with the expansion of definitions and uses the same refinement mappings leading to the same set of subgoals as before: (28) through (34).

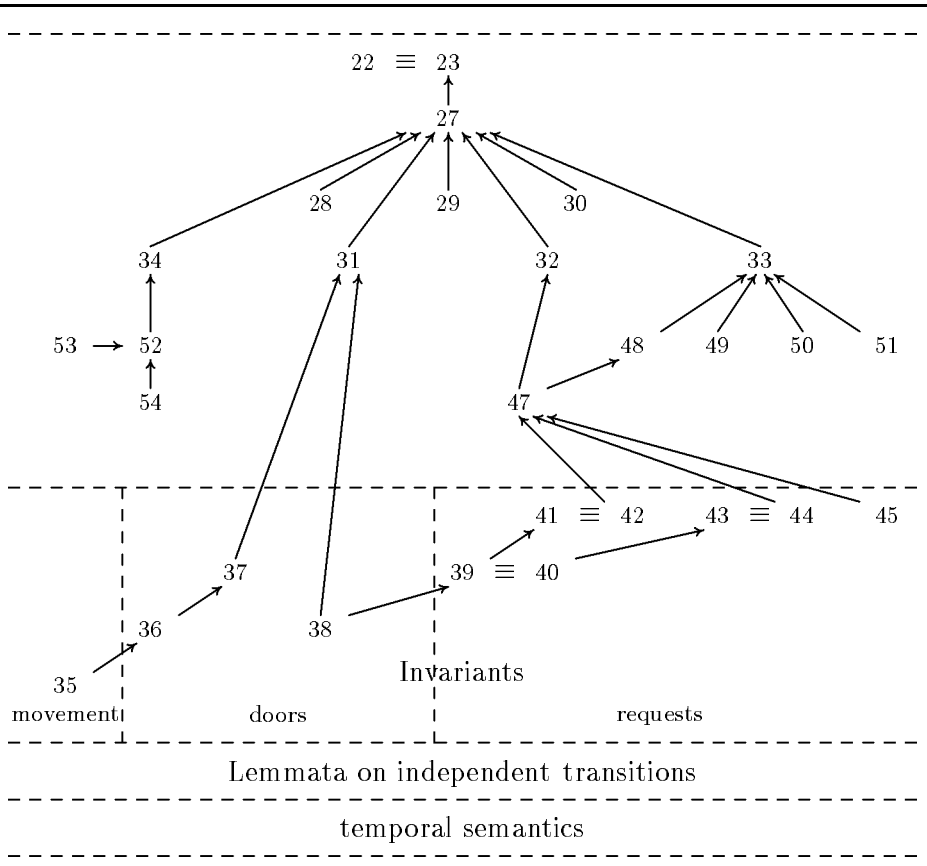


Figure 21: Structure of the proof of controller version 1

The difference in the proofs reflects the difference between the versions of the controller. The second one has to remember by itself for which of the floors there is a request. Using an additional invariant the requests of the plant and the controller can be related. Then the proofs of the subgoals are completely analogous, the additional invariant is used to prove correctness of the steps and that each request is eventually served.

To relate requests of the plant and the controller it is not necessary, that there is a one to one correspondence between the corresponding states of plant and controller. But no request of the plant should be lost by the controller and for each request recognized by the controller there should be a corresponding one of the plant. The difficult situation is, when the plant considers a request to be served by taking a transition from one of the states req to noreq, thereby generating one of the events served_*i*. This transition leads to a configuration in which the states lift.floor_*i*.request.noreq and control_2.req_*i* are active. The controller will change to the correct state in the next step, according to the presence of the event press_*i*. Remember, that the events press_*i* are discarded in the specification as long as the corresponding state lift.floor_*i*.request.req is active. Hence the invariant needed is

$$\begin{aligned} \Box\, in(\mathsf{control\_2.req\_}i) \;\Leftrightarrow\; & \vee\ in(\mathsf{lift.floor\_}i\mathsf{.request.req}) \\ & \vee\ \mathsf{served\_}i \in E_{exists} \end{aligned} \tag{55}$$

The invariant is valid initially and is preserved by stuttering steps and by steps introducing new external events. It remains to show

$$\begin{aligned} & \vee\ in(\mathsf{control\_2.req\_}i) \;\Leftrightarrow\; \vee\ in(\mathsf{lift.floor\_}i\mathsf{.request.req}) \\ & \qquad\qquad\qquad\qquad\ \vee\ \mathsf{served\_}i \in E_{exists} \\ & \vee\ step \\ & \Rightarrow in'(\mathsf{control\_2.req\_}i) \;\Leftrightarrow\; \vee\ in'(\mathsf{lift.floor\_}i\mathsf{.request.req}) \\ & \qquad\qquad\qquad\qquad\ \vee\ \mathsf{served\_}i \in E'_{exists} \end{aligned}$$

Proof: In case, that the state control_2.req_*i* is not active, $in(\mathsf{lift.floor\_}i\mathsf{.request.noreq})$ and $\mathsf{served\_}i \notin E_{exists}$ are valid. If $\mathsf{press\_}i \in E_{exists}$, then the plant as well as the controller recognize this request. If $\mathsf{press\_}i \notin E_{exists}$, then these states remain active and the event served_*i* cannot be generated by this step. Hence, the invariant is preserved by the step.

If the state control_2.req_*i* is active, either the state lift.floor_*i*.request.req is active or the event served_*i* is existing. Again the appropriate states, with respect to existence of the event press_*i*, are active after this step. ∎

8 Concluding remarks

In this paper a combination of a state-based formalism and a temporal logic has been proposed. It has been used to describe the environment – i.e. the plant –, the specification and the design – i.e. the controller – of a small reactive system. Thereby it has been shown, that this approach is feasible. In the different descriptions the state-based formalism and the temporal logic have been used to a different extent. The combination of the languages has been achieved by giving a temporal semantics of the state-based formalism in the temporal logic. Therefore, it is easily possible to relate the different descriptions formally, e.g. it has

been possible to prove a design correct with respect to a specification. Also, a specification can be considered as a refinement of the description of the plant.

When using the combined language to describe requirements, it is possible to express them in different specification styles. Some typical formulations of requirements have been shown in this paper. For some types of requirements one of the two constituents of the combined language can be used more conveniently than the other one. E.g. it has been appropriate to express invariance or liveness properties by a temporal formulae. On the other hand, it was easily possible to extend the description of the plant with additional states. It has to be investigated further, whether there are sufficient conditions which guarantee, that the specification of a requirement is a refinement of the description of the plant and that a specification is machine closed. It has to be considered also how the refinement relation can be maintained when combining several requirements.

The combined language will have to be used on more and larger examples to give more evidence to these observations. Furthermore, a more systematic approach to describe the plant and the requirements is necessary. It also has to be studied, whether it is possible to synthesize a controller given the description of the plant and the specification or whether there is a transformational approach deriving at least parts of the controller from the requirements.

Besides the use of the combined language on more realistic examples there is further work to be done with regard to the expressiveness of the language: The temporal semantics should be extended to capture data, a convenient notion of real-time, and cooperation between independent activities. Therefore, variables and actions of statecharts manipulating the data have to be incorporated in the semantics. Concerning real-time, it has to be investigated, what are the typical properties of a system, which have to be described, and how can they be expressed in the most convenient way. A good starting point to incorporate real-time in the semantics is [Abadi and Lamport, 1994]. Coordination between activities can be achieved by considering several activities as one large statechart, but also by extensions using explicit communication, see e.g. [Leveson *et al.*, 1994]. In this context it has also to be clarified how and to which extend the synchrony hypothesis should be supported.

Acknowledgments

I would like to thank Prof. Madlener for pointing out this direction of research and my colleagues M. Kronenburg and M. Schütze for valuable discussion on this topic.

References

[Abadi and Lamport, 1991] Martin Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.

[Abadi and Lamport, 1994] Martin Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM TOPLAS*, 16(5):1543–1571, September 1994.

[Alpern and Schneider, 1985] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[Antsaklis *et al.*, 1995] Panos Antsaklis, Wolf Kohn, Anil Nerode, and Shankar Sastry, editors. *Hybrid Systems II*, volume 999 of *LNCS*. Springer, 1995.

[Berry and Gonthier, 1992] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[Courtois and Parnas, 1993] P.-J. Courtois and David Lorge Parnas. Documentation for safety critical software. In *Proc. of 15th Intl. Conf. on Software Engineering*, pages 315–323, 1993.

[Day and Joyce, 1993] Nancy Day and Jeffrey J. Joyce. The semantics of statecharts in HOL. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Proc. of 6th Intl. Workshop on Higher Order Logic, Theorem Proving and its Applications*, volume 780 of *LNCS*, pages 338–351. Springer-Verlag, 1993.

[Day, 1993] Nancy Day. A model checker for statecharts (linking case tools with formal methods). Technical report 93-35, Department of Computer Science, Vancouver, B.C. Canada, October 1993.

[Grossmann *et al.*, 1993] Robert L. Grossmann, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors. *Hybrid Systems*, volume 736 of *LNCS*. Springer, 1993.

[Harel *et al.*, 1987] David Harel, Amir Pnueli, J.P. Schmidt, and Rivi Sherman. On the formal semantics of statecharts. In *Proc. of 2nd IEEE Symp. on Logic in Computer Science*, pages 54–64. IEEE Press, 1987.

[Harel, 1987] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Henzinger *et al.*, 1993] Thomas A. Henzinger, Zohar Manna, and Amir Pnueli. Towards refining temporal specifications into hybrid systems. In Robert L. Grossmann, Anil Nerode, Anders P. Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *LNCS*, pages 60–76. Springer, 1993.

[Hoare, 1985] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Kurshan, 1994] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton Series in Computer Science. Princeton University Press, 1994.

[Lamport, 1994a] Leslie Lamport. How to write a long formula. *Formal Aspects of Computing*, 6:580–584, 1994.

[Lamport, 1994b] Leslie Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, March 1994.

[Lamport, 1995] Leslie Lamport. TLA in pictures. *IEEE Transactions on Software Engineering*, 21(9):768–775, September 1995.

[Leveson *et al.*, 1994] Nancy G. Leveson, Mats Per Erik Heimdahl, Holly Hildreth, and Jon Damon Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[Manna and Pnueli, 1992] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems, Specification*. Springer, 1992.

[Manna and Pnueli, 1993] Zohar Manna and Amir Pnueli. Models for reactivity. *Acta Informatica*, 30:609–678, 1993.

[Milner, 1989] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[Olderog, 1991] Ernst-Rüdiger Olderog. *Nets, Terms and Formulas*. Number 23 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1991.

[Ostroff, 1989] Jonathan S. Ostroff. *Temporal Logic for Real-Time Systems*. Research Studies Press, 1989.

[Pnueli and Shalev, 1991] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A. R. Meyer, editors, *Intl. Conf. TACS '91: Theoretical Aspects of Computer Software*, volume 526 of *LNCS*, pages 244–264. Springer, 1991.

[Pnueli, 1992] Amir Pnueli. System specification and refinement in temporal logic. In R. Shyamasundar, editor, *12th Conf. on Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *LNCS*, pages 1–38. Springer, 1992.

[stm91, 1991] i-Logix, 22 Third Avenue, Burlington, Mass. 01803, USA. *STATEMATE, The Semantics of Statecharts*, January 1991.

[van der Beeck, 1994] Michael van der Beeck. A comparison of statechart variants. In H. Langmaack, W. P. de Roever, and J. Vytopil, editors, *Proc. of 3rd Intl. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *LNCS*, pages 128–148. Springer, 1994.

[van Schouwen *et al.*, 1993] A. John van Schouwen, David L. Parnas, and Jan Madey. Documentation of requirements for computer systems. In *Proc. of IEEE Intl. Symp. on Requirements Engineering*, pages 198–207. IEEE Computer Society Press, 1993.

[Zave and Yeh, 1981] Pamela Zave and Raymond T. Yeh. Executable requirements. In *Proc. 5th Intl. Conf. on Software Engineering*, pages 295–304. IEEE, 1981.

[Zave, 1991] Pamela Zave. An insider's evaluation of PAISLey. *IEEE Transactions on Software Engineering*, 17(3):212–225, March 1991.

List of Functions and Predicates

A Syntax of TLA

TLA formulae can be constructed as shown by the following syntax diagrams. These are taken from [Lamport, 1994b, pages 887 and 905]. See section 3 for an informal introduction of the semantics of TLA.

$\langle state\,function \rangle \triangleq$ non boolean expression containing constant symbols and variables

$\langle action \rangle \triangleq$ boolean−valued expression containaing constant symbols, variables, and primed variables

$\langle predicate \rangle \triangleq$
- $\langle action \rangle$ without primed variables
- $\mid$ $Enabled\,\langle action \rangle$

$\langle formula \rangle \triangleq$
- $\langle predicate \rangle$
- $\mid$ $\Box[\langle action \rangle]_{\langle state\,function \rangle}$
- $\mid$ $\neg\langle formula \rangle$
- $\mid$ $\langle formula \rangle \wedge \langle formula \rangle$
- $\mid$ $\Box\langle formula \rangle$

$\langle general\,formula \rangle \triangleq$
- $\langle formula \rangle$
- $\mid$ $\mathbf{\exists}\,\langle variable \rangle : \langle general\,formula \rangle$
- $\mid$ $\exists\,\langle rigid\,variable \rangle : general\,formula \rangle$
- $\mid$ $<< general\_formula >> \wedge << general\_formula >>$
- $\mid$ $<< general\_formula >>$

The following notation can be used also. Thereby f is a $\langle state\,function \rangle$, A is an $\langle action \rangle$, and F and G are $\langle formula \rangle$s.

$$
\begin{array}{lcl}
[A]_f & \triangleq & A \vee (f' = f) \\
\langle A \rangle_f & \triangleq & A \wedge (f' \neq f) \\
\Diamond F & \triangleq & \neg\Box\neg F \\
F \rightsquigarrow G & \triangleq & \Box(F \Rightarrow \Diamond G) \\
WF_f(A) & \triangleq & \Box\Diamond\langle A \rangle_f \vee \Box\Diamond\neg Enabled\,\langle A \rangle_f \\
SF_f(A) & \triangleq & \Box\Diamond\langle A \rangle_f \vee \Diamond\Box\neg Enabled\,\langle A \rangle_f
\end{array}
$$

B Proofrules of TLA

The proofrules of TLA are shown for easier reference, they are taken from [Lamport, 1994b, pages 888 and 905].

Let F, G, H_c be (general) TLA formulae, P, Q, I be predicates, A, B, N, M be actions, and f, g be state functions.

The rules of Simple Temporal Logic

STL1. $$\frac{F \text{ provable by propositional logic}}{\Box F}$$

STL2. $$\frac{}{\Box F \;\Rightarrow\; F}$$

STL3. $$\frac{}{\Box\Box F \;\equiv\; \Box F}$$

STL4. $$\frac{F \;\Rightarrow\; G}{\Box F \;\Rightarrow\; \Box G}$$

STL5. $$\frac{}{\Box(F \wedge G) \;\equiv\; (\Box F) \wedge (\Box G)}$$

STL6. $$\frac{}{(\Diamond\Box F) \wedge (\Diamond\Box G) \;\equiv\; \Diamond\Box(F \wedge G)}$$

LATTICE. Let $\succ$ be a well−founded partial order on a set S

$$\frac{\begin{array}{l} F \wedge (c \in S) \;\Rightarrow\; H_c \\ \qquad\qquad \rightsquigarrow \vee\; G \\ \qquad\qquad\qquad \vee\; \exists\, d \in S : (c \succ d) \wedge H_d \end{array}}{F \;\Rightarrow\; ((\exists\, c \in S : H_c) \rightsquigarrow G)}$$

The Basic Rules of TLA

TLA1. $$\frac{P \wedge (f' = f) \;\Rightarrow\; P'}{\Box P \;\equiv\; P \wedge \Box[P \;\Rightarrow\; P']_f}$$

TLA2. $$\frac{P \wedge [A]_f \;\Rightarrow\; Q \wedge [B]_g}{\Box P \wedge \Box[A]_f \;\Rightarrow\; \Box Q \wedge \Box[B]_g}$$

Additional rules

INV1. $$\frac{I \wedge [N]_f \;\Rightarrow\; I'}{I \wedge \Box[N]_f \;\Rightarrow\; \Box I}$$

INV2. $$\frac{}{\Box I \;\Rightarrow\; (\Box[N]_f \;\equiv\; \Box[N \wedge I \wedge I']_f)}$$

WF1. $$\frac{\begin{array}{l} P \wedge [N]_f \;\Rightarrow\; (P' \vee Q') \\ P \wedge <N \wedge A>_f \;\Rightarrow\; Q' \\ P \;\Rightarrow\; \mathit{Enabled} <A>_f \end{array}}{\Box[N]_f \wedge WF_f(A) \;\Rightarrow\; (P \rightsquigarrow Q)}$$

WF2. $$\frac{\begin{array}{l} < N \wedge B >_f \;\Rightarrow\; < \overline{M} >_{\overline{g}} \\ P \wedge P' \wedge < N \wedge A >_f \wedge \overline{Enabled < M >_g} \;\Rightarrow\; B \\ P \wedge \overline{Enabled < M >_g} \;\Rightarrow\; Enabled < A >_f \\ \Box[N \wedge \neg B]_f \wedge WF_f(A) \wedge \Box F \wedge \Diamond\Box \overline{Enabled < M >_g} \;\Rightarrow\; \Diamond\Box P \end{array}}{\Box[N]_f \wedge WF_f(A) \wedge \Box F \;\Rightarrow\; \overline{WF_g(M)}}$$

SF1. $$\frac{\begin{array}{l} P \wedge [N]_f \;\Rightarrow\; (P' \vee Q') \\ P \wedge < N \wedge A >_f \;\Rightarrow\; Q' \\ \Box P \wedge \Box[N]_f \wedge \Box F \;\Rightarrow\; \Diamond Enabled < A >_f \end{array}}{\Box[N]_f \wedge SF_f(A) \wedge \Box F \;\Rightarrow\; (P \rightsquigarrow Q)}$$

SF2. $$\frac{\begin{array}{l} < N \wedge B >_f \;\Rightarrow\; < \overline{M} >_{\overline{g}} \\ P \wedge P' \wedge < N \wedge A >_f \;\Rightarrow\; B \\ P \wedge \overline{Enabled < M >_g} \;\Rightarrow\; Enabled < A >_F \\ \Box[N \wedge \neg B]_f \wedge SF_f(A) \wedge \Box F \wedge \Box\Diamond \overline{Enabled < M >_g} \;\Rightarrow\; \Diamond\Box P \end{array}}{\Box[N]_f \wedge SF_f(A) \wedge \Box F \;\Rightarrow\; \overline{SF_g(M)}}$$

Quantification

Let x be a variable, c be a rigid variable, and e be a constant expression:

E1. $$\frac{}{F(f/x) \;\Rightarrow\; \exists x : F}$$

E2. $$\frac{F \;\Rightarrow\; G \qquad x \text{ does not occur free in } G}{(\exists x : F) \;\Rightarrow\; G}$$

F1. $$\frac{}{F(E/c) \;\Rightarrow\; \exists c : F}$$

F2 $$\frac{F \;\Rightarrow\; G \qquad c \text{ does not occur free in } G}{(\exists c : F) \;\Rightarrow\; G}$$